MATLAB®

App Building

MATLAB®

R2023a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

MATLAB® App Building

Trademarks

Patents

Revision History

November 2000	Online Only	New for MATLAB 6.0 (Release 12)
June 2001	Online Only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online Only	Revised for MATLAB 6.6 (Release 13)
June 2004	Online Only	Revised for MATLAB 7.0 (Release 14)
October 2004	Online Only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online Only	Revised for MATLAB 7.0.4 (Release 14SP2)
September 2005	Online Only	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Online Only	Revised for MATLAB 7.2 (Release 2006a)
May 2006	Online Only	Revised for MATLAB 7.2
September 2006	Online Only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online Only	Revised for MATLAB 7.4 (Release 2007a)
September 2007	Online Only	Revised for MATLAB 7.5 (Release 2007b)
March 2008	Online Only	Revised for MATLAB 7.6 (Release 2008a)
October 2008	Online Only	Revised for MATLAB 7.7 (Release 2008b)
March 2009	Online Only	Revised for MATLAB 7.8 (Release 2009a)
September 2009	Online Only	Revised for MATLAB 7.9 (Release 2009b)
March 2010	Online Only	Revised for MATLAB 7.10 (Release 2010a)
September 2010	Online Only	Revised for MATLAB 7.11 (Release 2010b)
April 2011	Online Only	Revised for MATLAB 7.12 (Release 2011a)
September 2011	Online Only	Revised for MATLAB 7.13 (Release 2011b)
March 2012	Online Only	Revised for MATLAB 7.14 (Release 2012a)
September 2012	Online Only	Revised for MATLAB 8.0 (Release 2012b)
March 2013	Online Only	Revised for MATLAB 8.1 (Release 2013a)
September 2013	Online Only	Revised for MATLAB 8.2 (Release 2013b)
March 2014	Online Only	Revised for MATLAB 8.3 (Release 2014a)
October 2014	Online Only	Revised for MATLAB 8.4 (Release 2014b)
March 2015	Online Only	Revised for MATLAB 8.5 (Release 2015a)
September 2015	Online Only	Revised for MATLAB 8.6 (Release 2015b)
March 2016	Online Only	Revised for MATLAB 9.0 (Release 2016a)
September 2016	Online Only	Revised for MATLAB 9.1 (Release 2016b)
March 2017	Online Only	Revised for MATLAB 9.2 (Release 2017a)
September 2017	Online Only	Revised for MATLAB 9.3 (Release 2017b)
March 2018	Online Only	Revised for MATLAB 9.4 (Release 2018a)
September 2018	Online Only	Revised for MATLAB 9.5 (Release 2018b)
March 2019	Online Only	Revised for MATLAB 9.6 (Release 2019a)
September 2019	Online Only	Revised for MATLAB 9.7 (Release 2019b)
March 2020	Online Only	Revised for MATLAB 9.8 (Release 2020a)
September 2020	Online Only	Revised for MATLAB 9.9 (Release 2020b)
March 2021	Online Only	Revised for MATLAB 9.10 (Release 2021a)
September 2021	Online Only	Revised for MATLAB 9.11 (Release 2021b)
March 2022	Online Only	Revised for MATLAB 9.12 (Release 2022a)
September 2022	Online Only	Revised for MATLAB 9.13 (Release 2022b)
March 2023	Online Only	Revised for MATLAB 9.14 (Release 2023a)

Contents

Introduction to Creating Apps

1 About Apps in MATLAB Software

2 How to Create a App with GUIDE

App Designer

3 App Designer Basics

4 Component Choices and Customizations

5 App Layout

6 App Programming

7 App Designer Examples

8 Advanced App Designer Examples

9 Keyboard Shortcuts

Create UIs Programmatically

10 Lay Out a Programmatic UI

11 Create and Manage Callbacks Programmatically

12 Developing Classes of UI Component Objects

13 Create Custom UI Components in App Designer

14 Transition or Maintain figure-Based Apps

15 Examples of Programmatic Apps

16 Live Editor Task Development

Create UIs with GUIDE

17 GUIDE Preferences and Options

18 Lay Out a UI Using GUIDE

19 Programming a GUIDE App

20 Examples of GUIDE UIs

App Packaging

21 Packaging GUIs as Apps

Introduction to Creating Apps

1

About Apps in MATLAB Software

Ways to Build Apps

You can use MATLAB to build interactive user interfaces that can be integrated into various environments. There are two types of user interfaces you can build:

- Apps — Self-contained interfaces that perform operations based on user interactions
- Live Editor tasks — Interfaces that can be embedded into a live script and that generate code as users explore parameters

The way that you build and share these interfaces, as well as the main file type for the interface, differs depending on the interface type. This table shows the differences.

Type	Ways to Build	File Type	Sharing Options
App	Interactively, using App Designer	`.mlapp`	• Distribute the main interface file and supporting files directly • Package as a single file • Deploy as a web app that can run in a web browser (requires MATLAB Compiler™) • Create a standalone desktop application (requires MATLAB Compiler)
	Programmatically, using MATLAB functions	`.m` (MATLAB script, function, or class file)	• Distribute the main interface file and supporting files directly • Package as a single file • Create a standalone desktop application (requires MATLAB Compiler)
Live Editor task	Programmatically, using the `matlab.task.LiveTask` base class	`.m` (MATLAB class file)	• Distribute the main interface file and supporting files directly

Build an App

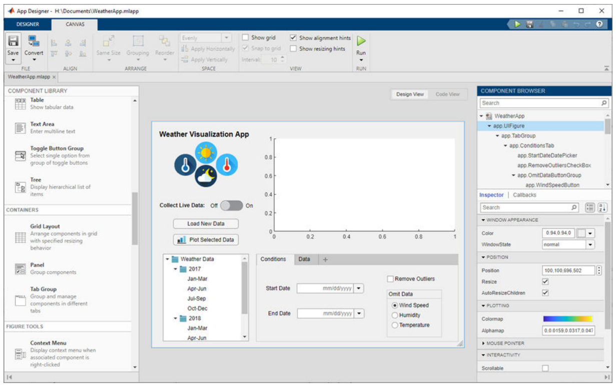
To create a self-contained user interface, build an app. You can build an app in multiple ways:

- Interactively, using App Designer
- Programmatically, using MATLAB functions

Each of these approaches offers a different workflow and a slightly different set of functionalities. The best choice for you depends on your project requirements and how you prefer to work.

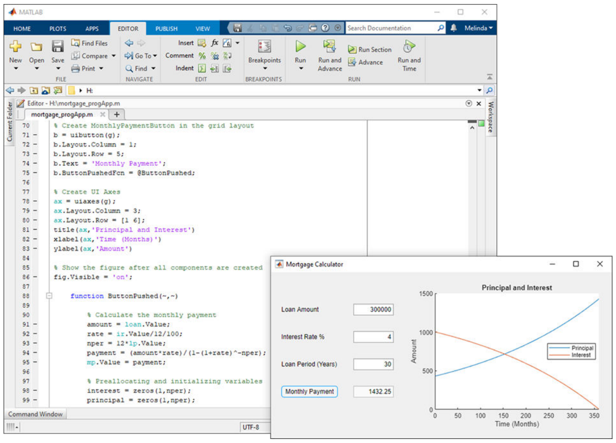
Use App Designer to Build Apps Interactively

App Designer is a rich interactive environment introduced in R2016a, and it is the recommended environment for building apps in MATLAB. It includes a fully integrated version of the MATLAB Editor. The layout design and code views are tightly linked so that changes you make in one view immediately affect the other. A larger set of interactive components is available, including date picker, tree, and image components. There are also features like a grid layout manager and automatic reflow options to make your app detect and adapt to changes in screen size. For more information, see “Develop Apps Using App Designer”.



Use MATLAB Functions to Build Apps Programmatically

You can also code the layout and behavior of your app entirely using MATLAB functions. In this approach, you create a figure to serve as the container for your UI by using either the `uifigure` or `figure` function. Then, you add components to it programmatically. Each type of figure supports different components and properties. The `uifigure` function is the recommended function for building new apps because it creates a figure that is specifically configured for app building. UI figures support the same types of modern graphics and interactive UI components that App Designer supports. For more information, see “Develop Apps Programmatically”.



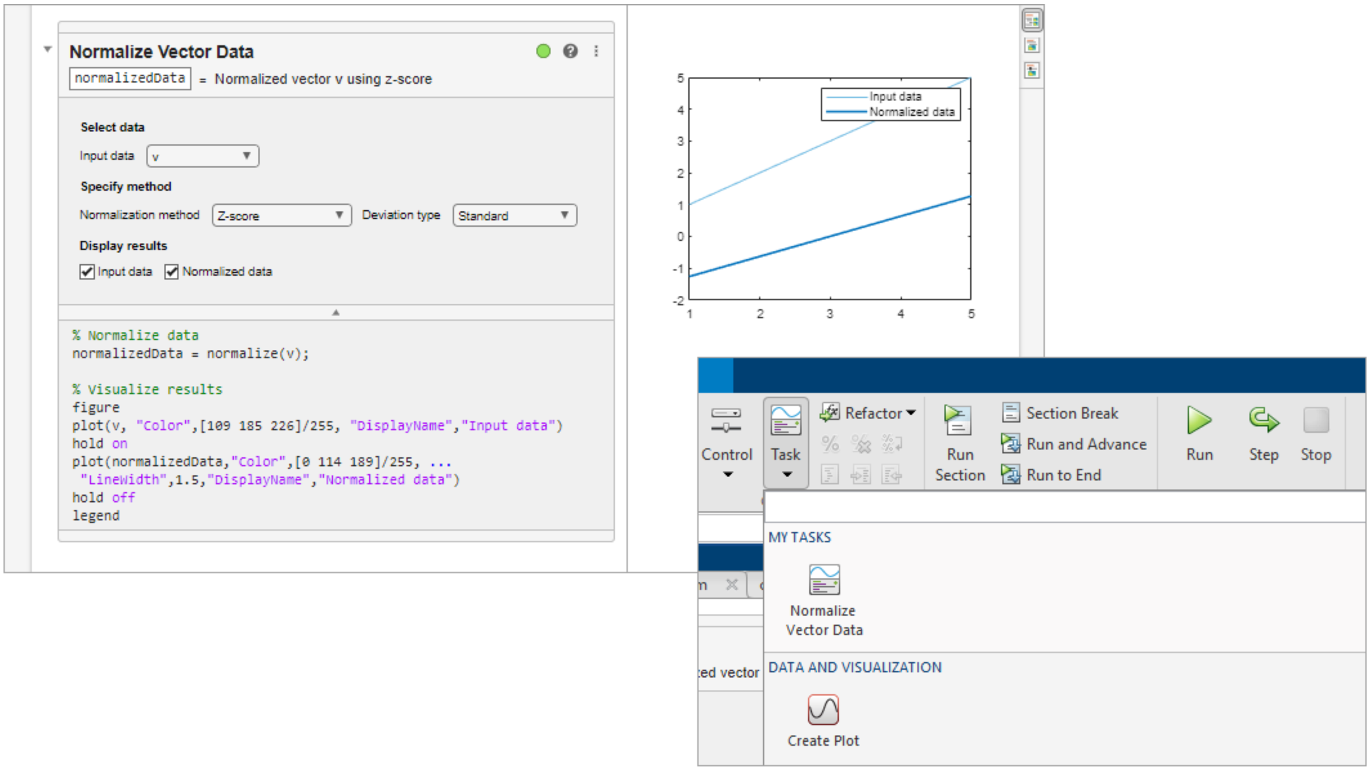
```
% Create MonthlyPaymentButton in the grid layout
b = uibutton(g);
b.Layout.Column = 1;
b.Layout.Row = 5;
b.Text = 'Monthly Payment';
b.ButtonPushedFcn = @ButtonPushed;

% Create UI Axes
ax = uiaxes(g);
ax.Layout.Column = 3;
ax.Layout.Row = [1 6];
title(ax,'Principal and Interest')
xlabel(ax,'Time (Months)')
ylabel(ax,'Amount')

% Show the figure after all components are created
fig.Visible = 'on';

    function ButtonPushed(~,~)

        % Calculate the monthly payment
        amount = loan.Value;
        rate = ir.Value/12/100;
        nper = 12*lp.Value;
        payment = (amount*rate)/(1-(1+rate)^-nper);
        mp.Value = payment;

        % Preallocating and initializing variables
        interest = zeros(1,nper);
        principal = zeros(1,nper);
```

Loan Amount 300000
Interest Rate % 4
Loan Period (Years) 30
Monthly Payment 1432.25

Build a Live Editor Task

To create an interface that can be embedded into a live script, build a Live Editor task. Live Editor tasks represent a series of MATLAB commands that are automatically generated as users explore parameters. Tasks are useful because they can help reduce development time, errors, and time spent plotting.

You can create a Live Editor task programmatically by defining a subclass of the `matlab.task.LiveTask` base class. Then, you programmatically add components to the task to configure the user interface, and you write code to generate the MATLAB commands and output for the task. For more information, see “Develop Live Editor Tasks”.



Normalize Vector Data

normalizedData = Normalized vector v using z-score

Select data

Input data: v

Specify method

Normalization method: Z-score   Deviation type: Standard

Display results

☑ Input data ☑ Normalized data

```
% Normalize data
normalizedData = normalize(v);

% Visualize results
figure
plot(v, "Color",[109 185 226]/255, "DisplayName","Input data")
hold on
plot(normalizedData,"Color",[0 114 189]/255, ...
 "LineWidth",1.5,"DisplayName","Normalized data")
hold off
legend
```

See Also

Related Examples

- “Create and Run a Simple App Using App Designer” on page 3-2
- “Create and Run a Simple Programmatic App” on page 15-2
- “Display Graphics in App Designer” on page 3-15
- “Update figure-Based Apps to Use uifigure” on page 14-2
- “GUIDE Migration Strategies” on page 3-7

How to Create a App with GUIDE

Files Generated by GUIDE

In this section...

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 3-7 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

Code Files and FIG-Files

By default, the first time you save or run your app, GUIDE save two files:

- A FIG-file, with extension `.fig`, that contains a complete description of the layout and each component, such as push buttons, axes, panels, menus, and so on. The FIG-file is a binary file and you cannot modify it except by changing the layout in GUIDE. FIG-files are specializations of MAT-files. See “Create Custom Programs to Read MAT-Files” for more information.
- A code file, with extension `.m`, that initially contains initialization code and templates for some callbacks that control behavior. You generally add callbacks you write for your components to this file. As the callbacks are functions, the code file can never be a MATLAB script.

  When you save your app for the first time, GUIDE automatically opens the code file in your default editor.

The FIG-file and the code file must have the same name. These two files usually reside in the same folder, and correspond to the tasks of laying out and programming the app. When you lay out the app in the Layout Editor, your components and layout are stored in the FIG-file. When you program the app, your code is stored in the corresponding code file.

Code File Structure

The code file that GUIDE generates is a function file. The name of the main function is the same as the name of the code file. For example, if the name of the code file is `mygui.m`, then the name of the main function is `mygui`. Each callback in the file is a local function of that main function.

When GUIDE generates a code file, it automatically includes templates for the most commonly used callbacks for each component. The code file also contains initialization code, as well as an opening function callback and an output function callback. It is your job to add code to the component callbacks for your app to work as you want. You can also add code to the opening function callback and the output function callback. The code file orders functions as shown in the following table.

Section	Description
Comments	Displayed at the command line in response to the `help` command.
Initialization	GUIDE initialization tasks. *Do not edit this code.*
Opening function	Performs your initialization tasks before the user has access to the UI.
Output function	Returns outputs to the MATLAB command line after the opening function returns control and before control returns to the command line.
Component and figure callbacks	Control the behavior of the window and of individual components. MATLAB software calls a callback in response to a particular event for a component or for the figure itself.
Utility/helper functions	Perform miscellaneous functions not directly associated with an event for the figure or a component.

Adding Callback Templates to an Existing Code File

When you save the app, GUIDE automatically adds templates for some callbacks to the code file. If you want to add other callbacks to the file, you can easily do so.

Within GUIDE, you can add a local callback function template to the code in any of the following ways. Select the component for which you want to add the callback, and then:

- Right-click the mouse button, and from the **View callbacks** submenu, select the desired callback.
- From **View > View Callbacks**, select the desired callback.
- Double-click a component to show its properties in the Property Inspector. In the Property Inspector, click the pencil-and-paper icon ✎ next to the name of the callback you want to install in the code file.
- For toolbar buttons, in the Toolbar Editor, click the **View** button next to **Clicked Callback** (for Push Tool buttons) or **On Callback**, or **Off Callback** (for Toggle Tools).

When you perform any of these actions, GUIDE adds the callback template to the code file, saves it, and opens it for editing at the callback you just added. If you select a callback that currently exists in the code file, GUIDE adds no callback, but saves the file and opens it for editing at the callback you select.

For more information, see “GUIDE-Generated Callback Functions and Property Values” on page 19-4.

About GUIDE-Generated Callbacks

Callbacks created by GUIDE for components are similar to callbacks created programmatically, with certain differences.

- GUIDE generates callbacks as function templates within the code file.

  GUIDE names callbacks based on the callback type and the component `Tag` property. For example, `togglebutton1_Callback` is such a default callback name. If you change a component `Tag`, GUIDE renames all its callbacks in the code file to contain the new tag. You can change the name of a callback, replace it with another function, or remove it entirely using the Property Inspector.

- GUIDE provides three arguments on page 19-4 to callbacks, always named the same.
- You can append arguments to GUIDE-generated callbacks, but never alter or remove the ones that GUIDE places there.
- You can rename a GUIDE-generated callback by editing its name or by changing the component `Tag`.
- You can delete a callback from a component by clearing it from the Property Inspector; this action does not remove anything from the code file.
- You can specify the same callback function for multiple components to enable them to share code.

After you delete a component in GUIDE, all callbacks it had remain in the code file. If you are sure that no other component uses the callbacks, you can then remove the callback code manually. For details, see “Renaming and Removing GUIDE-Generated Callbacks” on page 19-13.

See Also

Related Examples

- “Write Callbacks in GUIDE” on page 19-2

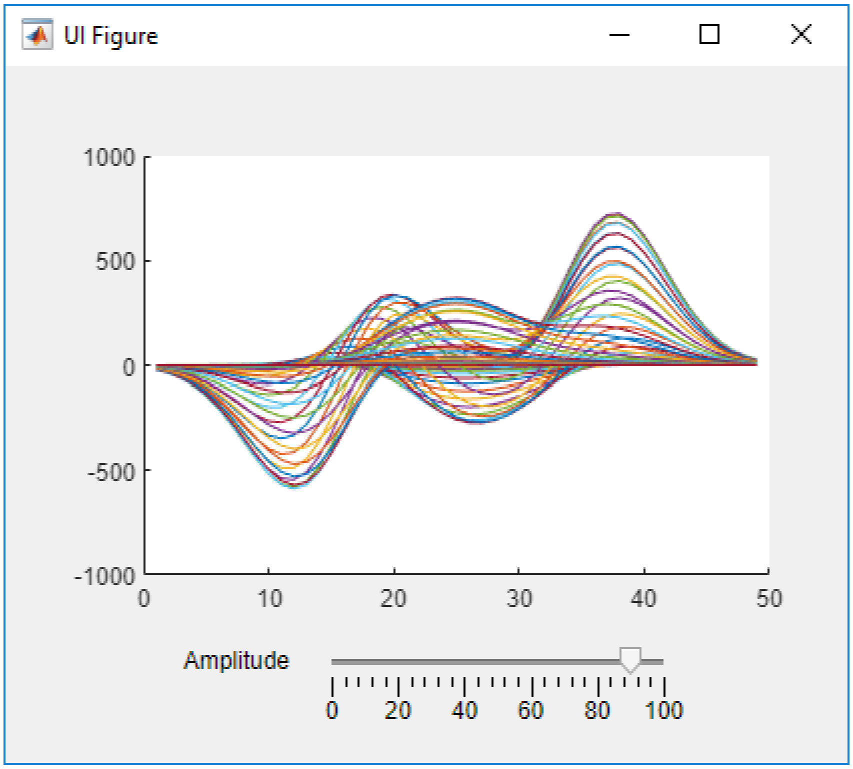
App Designer

3

App Designer Basics

Create and Run a Simple App Using App Designer

App Designer provides a tutorial that guides you through the process of creating a simple app containing a plot and a slider. The slider controls the amplitude of the plotted function. You can create this app by running the tutorial, or you can follow the tutorial steps listed here.

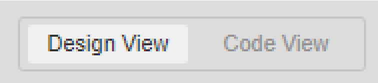


Run the Tutorial

To run the tutorial in App Designer, open the App Designer Start Page and click **Show examples** in the **Apps** section. Then, select **Interactive Tutorial**.

Tutorial Steps for Creating the App

App Designer has two views for creating an app: **Design View** and **Code View**. Use **Design View** to create UI components and interactively lay out your app. Use **Code View** to program your app behavior. You can switch between the two views using the toggle buttons in the upper right-corner of App Designer.



To create the simple plotting app, open a new app in App Designer and follow these steps.

Step 1: Create an Axes Component

In **Design View**, create UI components and modify their appearance interactively. The **Component Library** contains all components, containers, and tools that you can add to your app interactively. Add a component by dragging it from the **Component Library** onto the app canvas. You can then change the appearance of the component by setting properties in the **Component Browser**, or by editing certain aspects of the component, such as size and label text, directly on the canvas.

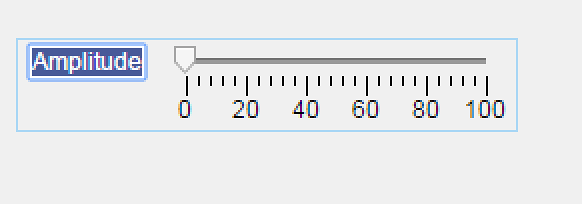
In your plotting app, create an axes component to display plotted data. Drag an **Axes** component from the **Component Library** onto the canvas.

Step 2: Create a Slider Component

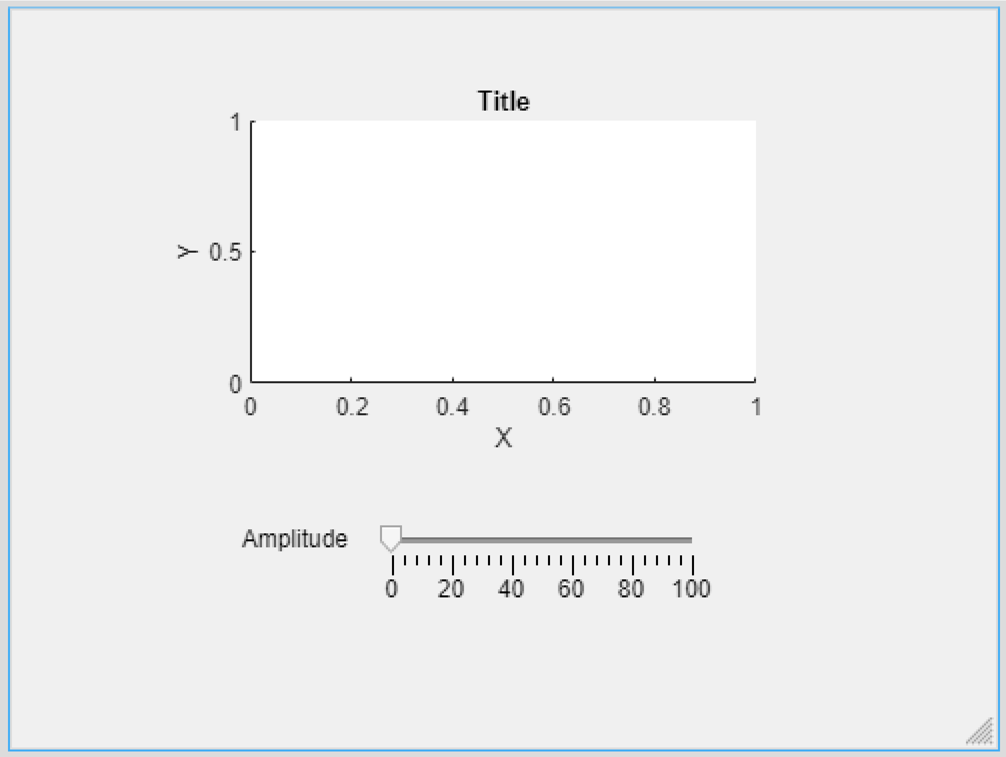
Drag a **Slider** component from the **Component Library** onto the canvas. Place it below the axes component.

Step 3: Update the Slider Label

Replace the slider label text. Double-click the label and replace the word `Slider` with `Amplitude`.



When you have finished laying out your app, the canvas in **Design View** should look like this:



For more information about laying out apps, see “Lay Out Apps in App Designer Design View” on page 5-2.

Step 4: Navigate to Code View

Once you have laid out your app, write code to program the behavior of your app. Click the **Code View** button above the canvas to edit your app code.

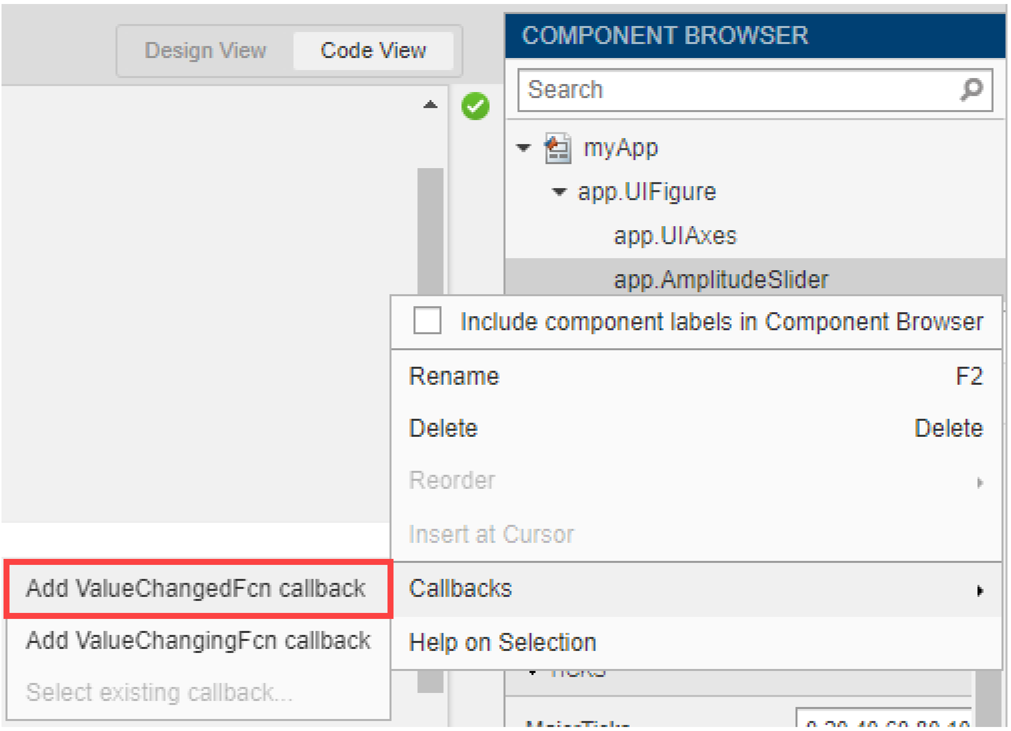
When you add components to your app in **Design View**, App Designer automatically generates code that executes when you run the app. This code configures your app appearance to match what you see on the canvas. This code is not editable and is displayed on a gray background. As part of this generated code, App Designer creates some objects for you to use when programming your app behavior.

- The `app` object — This object stores all of the data in your app, such as the UI components and any data you specify using properties. All functions in your app require this object as the first argument. This pattern enables you to have access to your components and properties from within those functions.
- The component objects — Whenever you add a component in **Design View**, App Designer stores the component as an object named using the form `app.`*`ComponentName`*. You can view and modify the names of the components in your app using the **Component Browser**. To access and update component properties from within your app code, use the pattern `app.`*`ComponentName`*`.`*`Property`*.

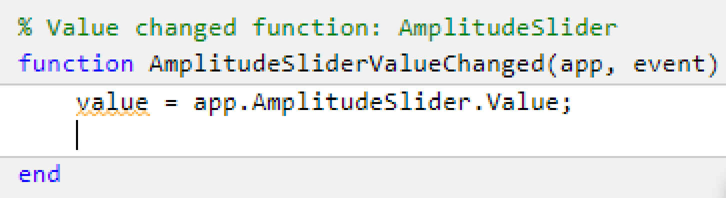
Step 5: Add a Slider Callback Function

Program your app behavior using callback functions. A callback function is a function that executes when the app user performs a specific interaction, such as adjusting the value of a slider.

In your plotting app, add a callback function that executes whenever the user adjusts the slider value. Right-click `app.AmplitudeSlider` in the **Component Browser**. Then select **Callbacks > Add ValueChangedFcn callback** in the context menu.



When you add a callback to a component, App Designer creates a callback function and places the cursor in the body of that function. App Designer automatically passes the `app` object as the first argument of the callback function to enable access components and their properties. For example, in the `AmplitudeSliderValueChanged` function, App Designer automatically generates a line of code to access the value of the slider.



```
% Value changed function: AmplitudeSlider
function AmplitudeSliderValueChanged(app, event)
    value = app.AmplitudeSlider.Value;

end
```

For more information about programming app behavior using callback functions, see “Callbacks in App Designer” on page 6-16.

Step 6: Plot Data

When you call a graphics function in App Designer, specify the target axes or parent object as an argument to the function.

In your plotting app, update the plotted data in the axes whenever the app user changes the slider value by specifying the name of the axes object in your app, `app.UIAxes`, as the first argument to the `plot` function. Add this code to the second line of the `AmplitudeSliderValueChanged` callback to plot the scaled output of the `peaks` function on the axes.

```
plot(app.UIAxes,value*peaks)
```

For more information about displaying graphics in an app, see “Display Graphics in App Designer” on page 3-15.

Step 7: Update Axes Limits

To access and update component properties from within your app code, use the pattern `app.`*`ComponentName`*`.`*`Property`*.

In your plotting app, change the limits of the *y*-axis by setting the `YLim` property of the `app.UIAxes` object. Add this command to the third line of the `AmplitudeSliderValueChanged` callback:

```
app.UIAxes.YLim = [-1000 1000];
```

Step 8: Run the App

Click ▶ **Run** to save and run the app. Adjust the value of the slider to plot some data in the app.

After saving your changes, your app is available for running again in App Designer or by typing its name (without the `.mlapp` extension) in the MATLAB Command Window. When you run the app from the command prompt, the file must be in the current folder or on the MATLAB path.

See Also

Related Examples

- “Lay Out Apps in App Designer Design View” on page 5-2
- “Manage Code in App Designer Code View” on page 6-2
- “Callbacks in App Designer” on page 6-16
- “Display Graphics in App Designer” on page 3-15
- “Share Data Within App Designer Apps” on page 6-26

GUIDE Migration Strategies

In R2019b, MathWorks® announced that GUIDE, the original drag-and-drop environment for building apps in MATLAB, will be removed in a future release. After GUIDE is removed, existing GUIDE apps (GUIs) will continue to run in MATLAB, and app program files will still be editable if you need to change the behavior of an app.

To continue editing the *layout* of an existing GUIDE app and help maintain its compatibility with future MATLAB releases, you must use one of the suggested migration strategies listed in this table.

App Development Needs	Migration Strategy	How to Migrate
Ongoing development	Migrate your app to App Designer.	Open the app in GUIDE and select **File > Migrate to App Designer**. In the GUIDE Transition Options dialog, click **Migrate**.
Occasional editing	Export your app to a single MATLAB file to manage your app layout and code using MATLAB functions.	Open the app in GUIDE and select **File > Export to MATLAB-file**. In the GUIDE Transition Options dialog, click **Export**.

Migrate GUIDE App to App Designer

Migrating your GUIDE app to App Designer allows you to continue developing the layout of your app interactively. It also allows you to take advantage of features like an enhanced UI component set and auto-reflow options to make your app responsive to changes in screen size. And it gives you the ability to create and share your app as a web app (requires MATLAB Compiler).

Use this option for GUIDE apps that require significant or ongoing feature development.

The GUIDE to App Designer Migration Tool for MATLAB was first released in R2018a to ease the conversion process. It is available through the Add-On Explorer in the MATLAB desktop or through File Exchange on MATLAB Central™.

Starting in R2020a, the migration tool has significant improvements that drastically reduce the time and the number of manual code updates required to get your app running in App Designer. For details about these enhancements, see “Callback Code” on page 3-8.

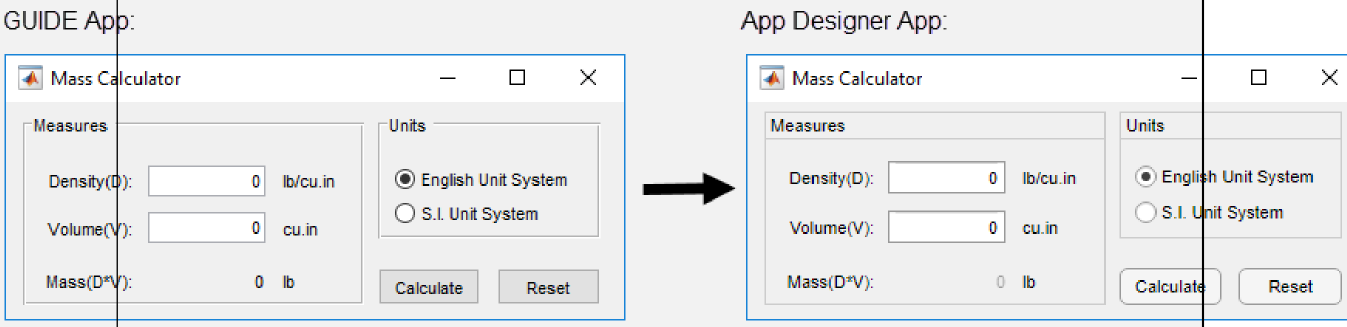
There are several ways to migrate your app, depending on which environment you begin in.

- In GUIDE, open your app and select **File > Migrate to App Designer**.
  - If you do not already have the GUIDE to App Designer Migration Tool installed, click **Install Support Package**. This opens the Add-On Explorer, where you can install the migration tool. Once you have installed the tool, reopen the GUIDE Transition Options dialog.
  - Once you have installed the GUIDE to App Designer Migration Tool, choose the correct FIG file and then click **Migrate**. The app migrates and automatically opens in App Designer.
- In App Designer, open any app and go to the **Designer** tab. In the **File** section, click **Open > Open GUIDE to App Designer Migration Tool**.

- In the MATLAB Command Window, call the `appmigration.migrateGUIDEApp` function. You can use this function to migrate multiple GUIDE apps as a batch.

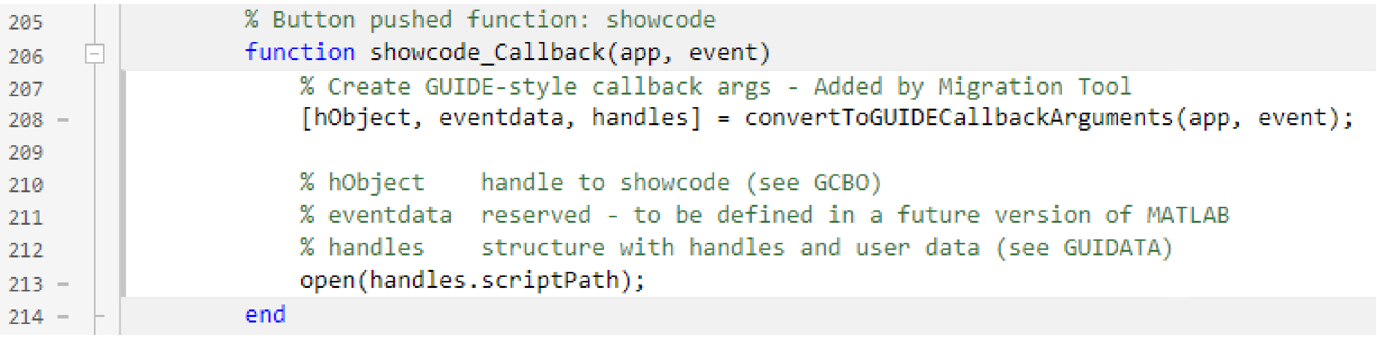
Features of the Migration Tool

The migration tool helps you convert your apps by reading in a GUIDE FIG file and automatically generating the App Designer equivalent components and layout in an MLAPP file. Your GUIDE callback code and other user-defined functions are copied into the MLAPP file. This semi-automated code conversion also creates a migration report that suggests actions for any manual code updates that are needed. Some features of the tool are described in this table.

Migration Tool Features	Description
File Conversion	Read in a GUIDE FIG file and associated code and then generate an App Designer MLAPP file. The App Designer file name takes the form `*guideFileName*_App.mlapp`.
Components and App Layout	Convert components and property configurations to App Designer equivalents, and preserve the layout of the app.  GUIDE App: App Designer App: Mass Calculator Measures Density(D): 0 lb/cu.in Volume(V): 0 cu.in Mass(D*V): 0 lb Units ● English Unit System ○ S.I. Unit System Calculate Reset
Callback Code	Retain a copy of the GUIDE callback code and user-defined functions in the MLAPP file.
Tutorial	Step through the changes made to your migrated app.
Migration Report	Summarize the actions successfully completed by the migration tool. List any limitations or unsupported functionality, specific to your app, with suggested actions if available.

Callback Code

In order to make your GUIDE-style callback code compatible with the App Designer UI components in your app, the migration tool uses a function called `convertToGUIDECallbackArguments`. This function converts App Designer callback arguments into the GUIDE-style callback arguments that your code requires. The `convertToGUIDECallbackArguments` function is added to the beginning of each migrated callback function. It takes the App Designer callback arguments `app` and `event` and returns the GUIDE-style callback arguments `hObject`, `eventdata`, and `handles`. For example:



```
        % Button pushed function: showcode
        function showcode_Callback(app, event)
            % Create GUIDE-style callback args - Added by Migration Tool
            [hObject, eventdata, handles] = convertToGUIDECallbackArguments(app, event);

            % hObject    handle to showcode (see GCBO)
            % eventdata  reserved - to be defined in a future version of MATLAB
            % handles    structure with handles and user data (see GUIDATA)
            open(handles.scriptPath);
        end
```

Each of the GUIDE-style callback arguments is used for a different purpose:

- `hObject` is the handle of the object whose callback is executing. For components from your GUIDE app that were `UIControl` or `ButtonGroup` objects, `hObject` is a handle to a `UIControlPropertiesConverter` or `ButtonGroupPropertiesConverter` object. These objects are created to make your GUIDE-style code work in your App Designer callback functions.
- `eventdata` is usually empty, but can be a structure containing specific information about the callback event.
- `handles` is a structure that contains the migrated child components of the UI figure that have a `'Tag'` property value set. Child components that were `UIControl` objects in your GUIDE app are `UIControlPropertiesConverter` objects in the migrated app. Similarly, child `ButtonGroup` objects are `ButtonGroupPropertiesConverter` objects in the migrated app.

The `UIControlPropertiesConverter` and `ButtonGroupPropertiesConverter` objects act like adapters between the GUIDE-style code and the App Designer components and callbacks. A `UIControlPropertiesConverter` object is created for each component in your GUIDE app that was a `UIControl` object. These converter objects are associated with an App Designer UI component in your migrated app. The converter object has the same properties and values as the original `UIControl` from your GUIDE app, but it applies them to its associated App Designer UI component.

Similarly, for `ButtonGroup` objects from GUIDE, a `ButtonGroupPropertiesConverter` object is created in App Designer. This object makes it possible to set the `SelectedObject` property to a `UIControlPropertiesConverter` object so that button group `SelectionChangedFcn` callback logic will function.

Special Considerations

There are some circumstances that require you to take extra steps before or after you migrate your app. This table lists common scenarios and coding patterns that require extra steps or manual code updates. This is not intended to be a comprehensive list.

GUIDE App Feature	Description	Suggested Actions
Multiwindow apps (that is, two or more apps that share data)	Multiwindow apps require each app to be migrated separately. Migrated app file names are appended with _App. Calls to these apps from other apps must be updated.	Migrate each app separately. In the calling app, update the name of the app that is being called to the new file name.

GUIDE App Feature	Description	Suggested Actions
Radio buttons and radio button callbacks	The migration tool does not migrate radio buttons that are not parented to a radio button group, or callback functions for individual radio buttons.	Create a button group in App Designer and add radio buttons to it. To execute behavior when radio button selection is changed, create a `SelectionChangedFcn` callback function for the button group. For more information, see `uiradiobutton` and ButtonGroup Properties.
`uistack`	Calling this function in App Designer is not supported.	Determine if this functionality is critical to your app before migrating. There is no workaround in App Designer.
`findobj`, `findall`, and `gcbo`	Using `findobj`, `findall`, or `gcbo` to reference components and set properties can error. `UIControl` objects are migrated to the equivalent App Designer UI component. To access and set properties on these migrated components, you must set it on the `UIControlPropertiesConverter` objects. Or, you can update your code to use its associated App Designer component, properties, and values.	Reference components using the `handles` structure instead, or update your code to use the associated App Designer component, properties, and values.
`nargin` and `nargchk`	Helper functions are migrated to app methods and have `app` as an additional input argument. This can cause incorrect `nargin` or `nargchk` logic.	Increment check values by `1`.

GUIDE App Feature	Description	Suggested Actions

`OutputFcn(varargout)` and `Figure` output	There is no equivalent functionality in App Designer.
	When you instantiate a migrated App Designer app, the output is always the app object, not the `Figure` object.
		If your `OutputFcn` function includes initialization code that is critical to your app, then add it to the end of the `OpeningFcn` instead.
		If your `OutputFcn` function specifies output to be assigned to the workspace when you instantiate the app, such as the `Figure` object, then you need to create a function that instantiates the app. For example:
		`function out = MyGUIDEApp(varargin)`
		`    app = MyMigratedApp(varargin{:});`
		`    out = app.UIFigure;`
		`end`

If your GUIDE app integrates third-party components using functions like `actxcontrol`, see Recommendations for MATLAB Apps Using Java and ActiveX.

Aids for Adding New Features or Fully Adopting App Designer Code Style

App Designer and GUIDE have different code structures, callback syntaxes, and techniques for accessing UI components and sharing data. Understanding these differences is useful if you plan to add new App Designer features to your migrated app or want to update it to use App Designer code style and conventions. This table summarizes some of these differences.

Difference	GUIDE	App Designer	More Information
Using Figures and Graphics	GUIDE calls the `figure` function to create the app window. GUIDE calls the `axes` function to create axes for displaying plots. All MATLAB graphics functions are supported. There is no need to specify the target axes.	App Designer calls the `uifigure` function to create the app window. App Designer calls the `uiaxes` function to create axes for displaying plots. Most MATLAB graphics functions are supported.	“Display Graphics in App Designer” on page 3-15

Difference	GUIDE	App Designer	More Information
Using Components	GUIDE creates most components with the uicontrol function. Fewer components are available.	App Designer creates each UI component with its own dedicated function. More components are available, including Tree, Gauge, TabGroup, and DatePicker.	"App Building Components" on page 4-2 "Update UIControl Objects and Callbacks" on page 14-11
Accessing Component Properties	GUIDE uses set and get to access component properties, and uses handles to specify a component. For example, name = get(handles.Fig,'Name')	App Designer supports set and get, but encourages the use of dot notation to access component properties, and uses app to specify a component. For example, name = app.UIFigure.Name	"Callbacks in App Designer" on page 6-16
Managing App Code	The code is defined as a main function that can call local functions. All code is editable.	The code is defined as a MATLAB class. Only callbacks, helper functions, and custom properties are editable.	"Manage Code in App Designer Code View" on page 6-2
Writing Callbacks	Required callback input arguments are handles, hObject, and eventdata. For example, myCallback(hObject,eventdata,handles)	Required callback input arguments are app and event. For example, myCallback(app,event)	"Callbacks in App Designer" on page 6-16
Sharing Data	To store and share data between callbacks and functions, use the UserData property, the handles structure, or the guidata, setappdata, or getappdata function. For example, handles.currSelection = selection; guidata(hObject,handles);	To store and share data between callbacks and functions, use custom properties to create variables. For example, app.currSelection = selection	"Share Data Within App Designer Apps" on page 6-26

If you want to update the callback code in your migrated app to use App Designer code style and conventions, follow these steps:

1. In your callback functions, update references to the handles structure to instead use the app object. The handles structure gives access to converter objects that represent UIControl objects in your GUIDE app, whereas the app object gives access to the UI components in the App Designer app.

For example, a GUIDE-style callback sets the `BackgroundColor` of a push button style `UIControl` object using this code:

```
handles.pushbutton1.BackgroundColor = 'red';
```

Update this code to set the button UI component background color directly:

```
app.pushbutton1.BackgroundColor = 'red';
```

2. Update the properties that your callback code sets. In general, `UIControl` objects and their equivalent UI component objects have many of the same properties. However, there are some differences in the property names or the types of values that the properties accept. To see a comparison between `UIControl` and UI component objects and properties, and to learn how to update your code to use UI components, see “Update UIControl Objects and Callbacks” on page 14-11.
3. Once a callback function does not use the `hObject`, `eventdata`, or `handles` arguments, delete the line of code added by the Migration Tool that creates those arguments:

```
[hObject,eventdata,handles] = convertToGUIDECallbackArguments(app,event);
```

If your app creates dialog boxes using functions such as `errordlg` or `warndlg`, you can also update your code to take advantage of modern dialog boxes created specifically for app building, such as `uialert` and `uiconfirm`. For more information, see “Update Dialog Boxes” on page 14-18.

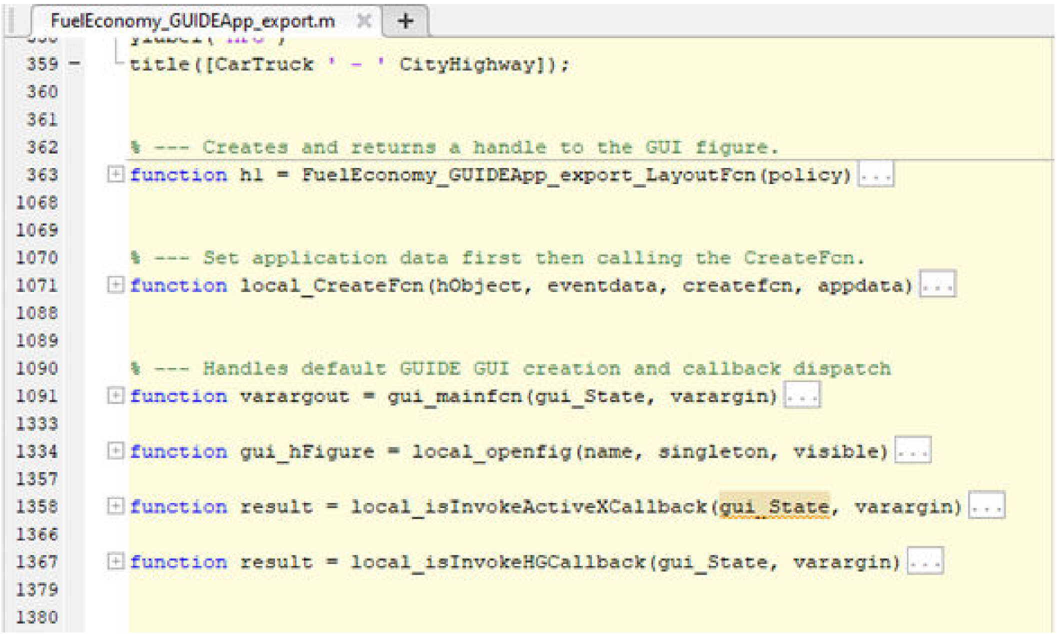
Export GUIDE App to MATLAB File

Exporting a GUIDE app converts it into a programmatic app by recreating the GUIDE FIG and program files together in a single MATLAB program file.

Use this option if you plan to:

- Make minor changes to the layout or behavior of your app.
- Develop your app programmatically, not interactively.

To export your app, open it in GUIDE and select **File** > **Export to MATLAB-file**, or right-click the FIG file in the MATLAB **Current Folder** browser and select **Export to MATLAB-file**. This brings up the GUIDE Transition Options dialog box. Verify that the correct FIG file is selected and then click **Export**. MATLAB creates a program file with `_export` appended to the file name. The new file contains your original callback code plus auto-generated functions that handle the creation and layout of the app. An example of these added functions is shown here.



```
title([CarTruck ' - ' CityHighway]);


% --- Creates and returns a handle to the GUI figure.
function h1 = FuelEconomy_GUIDEApp_export_LayoutFcn(policy) ...


% --- Set application data first then calling the CreateFcn.
function local_CreateFcn(hObject, eventdata, createfcn, appdata) ...


% --- Handles default GUIDE GUI creation and callback dispatch
function varargout = gui_mainfcn(gui_State, varargin) ...

function gui_hFigure = local_openfig(name, singleton, visible) ...

function result = local_isInvokeActiveXCallback(gui_State, varargin) ...

function result = local_isInvokeHGCallback(gui_State, varargin) ...

```

See Also

`appmigration.migrateGUIDEApp`

Related Examples

- “Create and Run a Simple App Using App Designer” on page 3-2
- “Display Graphics in App Designer” on page 3-15
- “Ways to Build Apps” on page 1-2

Display Graphics in App Designer

In this section...

App Designer Graphics Overview

Many of the graphics functions in MATLAB (and MATLAB toolboxes) have an argument for specifying the target axes or parent object. This argument is optional in most contexts, but when you call these functions in App Designer, you must specify this argument. The reason is that, in most contexts, MATLAB defaults to using the `gcf` or `gca` functions to get the target object for an operation. But these functions depend on the `HandleVisibility` property of the parent figure being `'on'`, and the `HandleVisibility` property of App Designer figures is set to `'off'` by default. This means that `gcf` and `gca` do not work as normal. As a result, omitting the argument for a target axes or parent object can produce unexpected results.

Depending on the graphics function you call, you might need to specify:

- A `UIAxes` component on the canvas
- A parent container in your app
- An axes component that you create programmatically in your app code

There are a number of ways to specify the target component for a graphics function. Some examples of the most common syntaxes are given below. To determine the correct target and syntax in your context, see the documentation for the specific graphics function you are using.

Display Graphics on Existing Axes

The most common way to display graphics in App Designer is to specify a `UIAxes` object on the App Designer canvas as the graphics function target. When you drag an axes component from the **Component Library** onto the canvas, this creates a `UIAxes` object in your app. The default name for an App Designer axes object is `app.UIAxes`. To determine or change the name of a specific axes on your canvas, select the axes component. Its name is listed and can be edited in the **Component Browser**

Specify Axes as First Argument

Many graphics functions have an optional first input argument to specify the target axes object. For example, both the `plot` function and the `hold` function take a target axes object in this way. To plot two lines on a set of axes on the canvas, specify the name of the axes object as the first argument to each function you call.

```
plot(app.UIAxes,[1 2 3 4],'-r');
hold(app.UIAxes);
plot(app.UIAxes,[10 9 4 7],'--b');
```

Specify Axes as Name-Value Argument

Some graphics functions require the target axes object to be specified as a name-value argument. For example, when you call the `imshow` and `triplot` functions, specify the axes object to display on using the `'Parent'` name-value argument. This code displays an image on an existing set of axes on your canvas:

```
imshow('peppers.png','Parent',app.UIAxes);
```

Display Graphics in Container

Some graphics functions display in a container component, such as a figure, panel, or grid layout, instead of an axes object. For example, the `heatmap` function has an optional first argument for specifying the container that the chart will display in.

Every App Designer app has a figure object, by default named `app.UIFigure`, that is a container for the components that make up the main app window. Specify `app.UIFigure` as the parent container argument to display graphics in the main app window. For example, to create a heat map in your app, use this syntax:

```
h = heatmap(app.UIFigure,rand(10));
```

To further organize and compartmentalize graphics that take a parent container input argument, drag a container component such as a panel, tab, or grid layout from the **Component Library** onto the canvas. Determine the name of the component by selecting it and viewing its name in the **Component Browser**. You can then specify this container as the parent when you call the graphics function.

Other commonly used graphics functions that take a parent container as input include `annotation`, `geobubble`, `parallelplot`, `scatterhistogram`, `stackedplot`, and `wordcloud`.

Create Axes Programmatically

Some graphics functions plot data on specialized axes. For example, functions that plot polar data must do so on a `PolarAxes` object. Unlike `UIAxes` objects, which you can add to your app from the **Component Library**, you must add specialized axes to your app *programmatically* in your code. To create an axes object programmatically, create a `StartupFcn` callback for your app. Within it, call the appropriate graphics function and specify a parent container in your app as the target.

Plot on Polar Axes

Functions such as `polarplot`, `polarhistogram`, and `polarscatter` take a polar axes object as a target. Create a polar axes programmatically by calling the `polaraxes` function. For example, to plot a polar equation in a panel, first drag a panel component from the **Component Library** onto your canvas. In the code for your app, create the polar axes object by calling the `polaraxes` function and specifying the panel as the parent container. Then, plot your equation with the `polarplot` function, specifying the polar axes as the target axes.

```
theta = 0:0.01:2*pi;
rho = sin(2*theta).*cos(2*theta);
```

```
pax = polaraxes(app.Panel);
polarplot(pax,theta,rho)
```

Plot on Geographic Axes

Functions such as `geoplot`, `geoscatter`, and `geodensityplot` take a geographic axes object as a target. Create a geographic axes programmatically by calling the `geoaxes` function. For example, to plot geographic data in a panel, use the following code:

```
latSeattle = 47 + 37/60;
lonSeattle = -(122 + 20/60);
gx = geoaxes(app.Panel);
geoplot(gx,latSeattle,lonSeattle)
```

Create Tiled Chart Layout

To tile multiple charts using the `tiledlayout` function, create a tiled chart layout in a panel and programmatically create axes in it using the `nexttile` function. Return the axes object from the `nexttile` function and use it to specify the axes for your charts or plots.

```
t = tiledlayout(app.Panel,2,1);
[X,Y,Z] = peaks(20)

% Tile 1
ax1 = nexttile(t);
surf(ax1,X,Y,Z)

% Tile 2
ax2 = nexttile(t);
contour(ax2,X,Y,Z)
```

Use Functions with No Target Argument

Some graphics functions, such as `ginput` and `gtext`, do not have an argument for specifying a target. As a result, you must set the `HandleVisibility` property of the App Designer figure to `'callback'` or `'on'` before calling these functions. After you call these functions, you can set the `HandleVisibility` property back to `'off'`. For example, this code shows how to define a callback that allows you to identify the coordinates of two points using the `ginput` function.

```
function pushButtonCallback(app,event)
    app.UIFigure.HandleVisibility = 'callback';
    ginput(2)
    app.UIFigure.HandleVisibility = 'off';
end
```

Use Functions That Don't Support Automatic Resizing

App Designer figures are resizable by default. This means that when you run an app and resize the figure window, components in the figure are automatically resized and repositioned to fit. However, some graphics functions do not support automatic resizing. To use these functions in App Designer, create a panel in which to display the output of the function and set the `AutoResizeChildren` property of the panel to `'off'`. You can set this property in the **Panel** tab of the **Component Browser** or in your code.

For example, the `subplot` function does not support automatic resizing. To use this function in your app:

1. Drag a panel component from the **Component Library** onto your canvas.
2. Set the AutoResizeChildren property of the panel to 'off'.
3. Specify the panel as the parent container using the 'Parent' name-value argument when you call subplot. Also, specify an output argument to store the axes.
4. Call the plotting function with the axes as the first input argument.

```
app.Panel.AutoResizeChildren = 'off';
ax1 = subplot(1,2,1,'Parent',app.Panel);
ax2 = subplot(1,2,2,'Parent',app.Panel);
plot(ax1,[1 2 3 4])
plot(ax2,[10 9 4 7])
```

Other commonly used functions that do not support automatic resizing include pareto and plotmatrix.

For more information about managing resize behavior, see “Alternatives to Default Auto-Resize Behaviors” on page 5-12.

Unsupported Functionality

Some graphics functionality is not supported in App Designer. This table lists the unsupported functionality that is most relevant to app building workflows.

Category	Not Supported
Retrieving and Saving Data	These functions are not supported: hgexport, hgload, hgsave, save, load, savefig, openfig, and saveas, and print. Instead of the saveas or print functions, use the exportapp function to save the content of an app window. To save plots in an app, use the exportgraphics or copygraphics functions. Figures created programmatically with uifigure do support the save, load, savefig, and openfig functions.
Web Apps	If you are using App Designer to create a deployed web app (requires MATLAB Compiler), additional graphics limitations apply. For more information, see “Web App Limitations and Unsupported Functionality” (MATLAB Compiler).

See Also

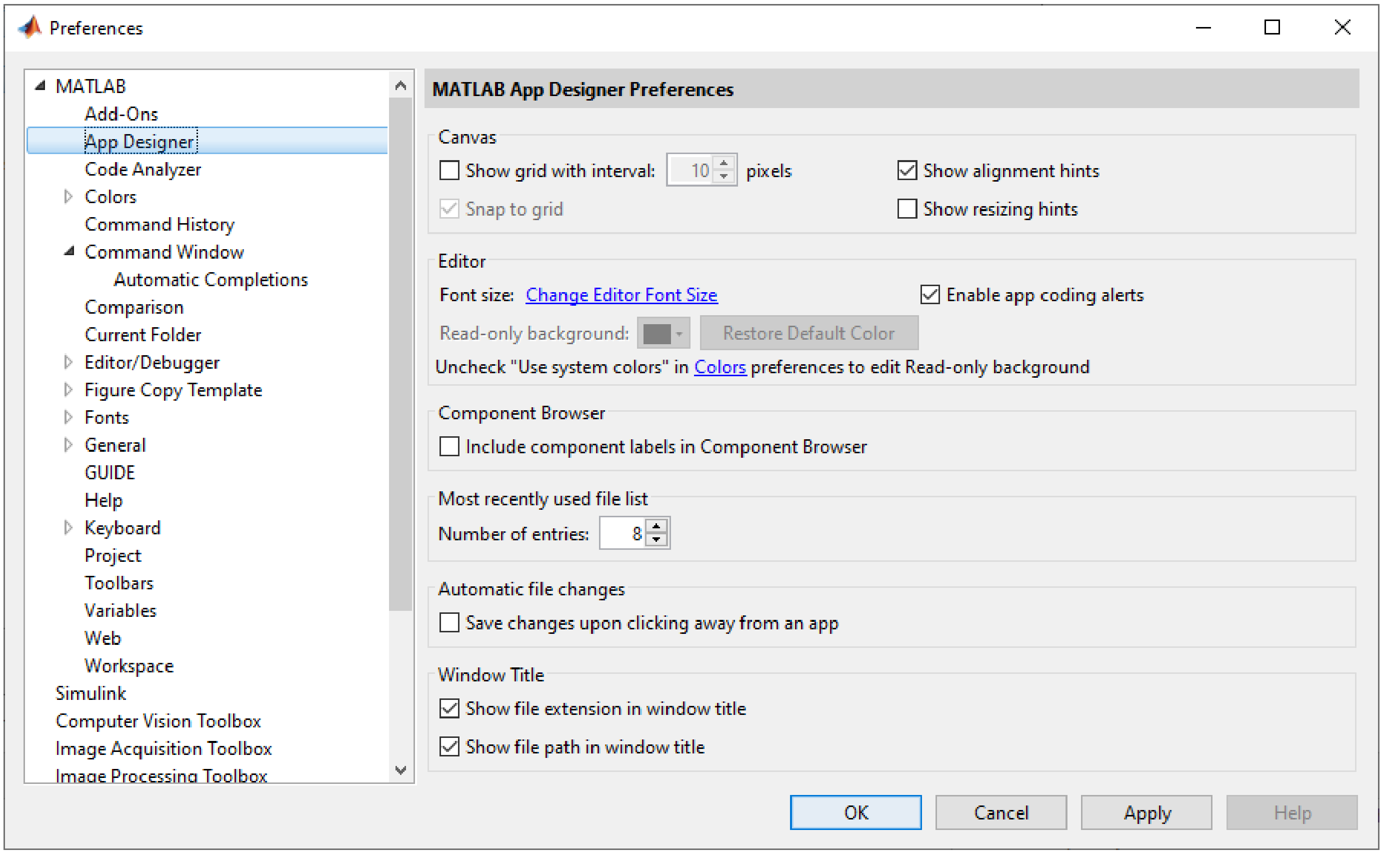
UI Figure | UIAxes

More About

- “Create Polar Axes Programmatically in an App” on page 7-7
- “App Building Components” on page 4-2
- “Add UI Components to App Designer Programmatically” on page 4-20
- “Manage Resizable Apps in App Designer” on page 5-12

App Designer Preferences

You can set App Designer preferences in the MATLAB Preferences dialog box. To open the dialog box, click ⚙ **Preferences** in the MATLAB Toolstrip. Then, select App Designer in the left pane.



MATLAB App Designer Preferences

Canvas

□ Show grid with interval: 10 pixels ☑ Show alignment hints

☑ Snap to grid □ Show resizing hints

Editor

Font size: Change Editor Font Size ☑ Enable app coding alerts

Read-only background: ▨ Restore Default Color

Uncheck "Use system colors" in Colors preferences to edit Read-only background

Component Browser

□ Include component labels in Component Browser

Most recently used file list

Number of entries: 8

Automatic file changes

□ Save changes upon clicking away from an app

Window Title

☑ Show file extension in window title

☑ Show file path in window title

This table describes each option in the right pane.

Option	Description
Show grid with interval	When selected, App Designer overlays a grid onto the canvas as an alignment aide. You can change the grid spacing to a specific number of pixels. The default spacing is 10.
Snap to grid	When selected, the upper left corner of a component always snaps to the intersection of two grid lines whenever you resize or move the component on the canvas.
Show alignment hints	When selected, App Designer displays alignment hints as you resize or move a component on the canvas.
Show resizing hints	When selected, App Designer displays the size of a component as you resize it on the canvas.

Option	Description
Font size	To change the font size that displays in App Designer **Code View**, click the link and modify the **Desktop code font** size. The font size can range from 8–48. The default font size is 10.
Enable app coding alerts	When selected, App Designer flags coding problems in the editor as you write code.
Read-only background	You can change the background color of the uneditable code sections in App Designer **Code View**. To change the background color, clear the **Use system colors** check box in the MATLAB Colors preferences. Then, select a new color from the color drop-down in the App Designer preferences. The default background color is gray.
Include component labels in Component Browser	When selected, labels included with components (such as edit fields) appear as separate items in the **Component Browser**. When this item is not selected, those labels do not appear in the **Component Browser**.
Number of entries (most recently used file list)	This number specifies how many of the most recently accessed apps appear under the **Recent Files** section of the **Open** menu in the **Designer** tab.
Save changes upon clicking away from an app	When selected, App Designer automatically saves changes to an app when you click away from it to switch between apps or to bring another window into focus. If an app has not already been saved at least once, autosave has no effect.
Show file extension in window title	When selected, App Designer displays the file extension of the active app in the App Designer window title.
Show file path in window title	When selected, App Designer displays the full path to the active app in the App Designer window title. When this item is not selected, App Designer displays only the app file name.

To customize the App Designer canvas and **Component Browser** settings programmatically, use `matlab.appdesigner Settings`.

See Also

Related Examples

- “Lay Out Apps in App Designer Design View” on page 5-2
- “Manage Code in App Designer Code View” on page 6-2

Component Choices and Customizations

App Building Components

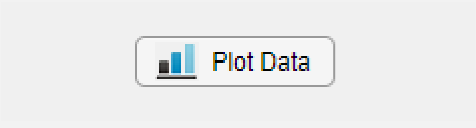
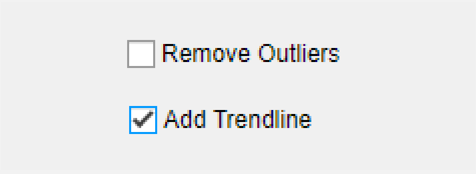
App Designer and UI figures support a large set of components for designing modern, full-featured applications. The tables below list the components that are available.

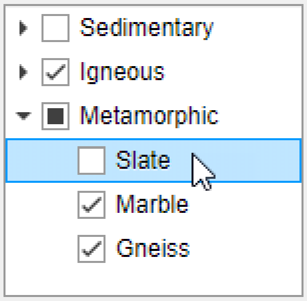
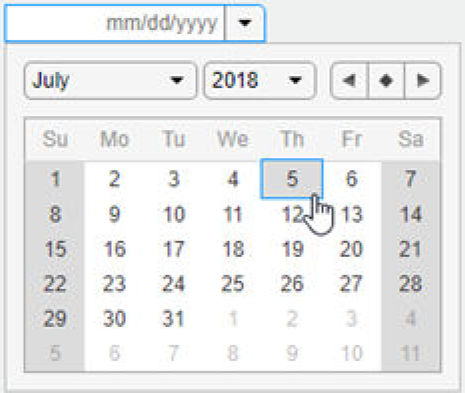
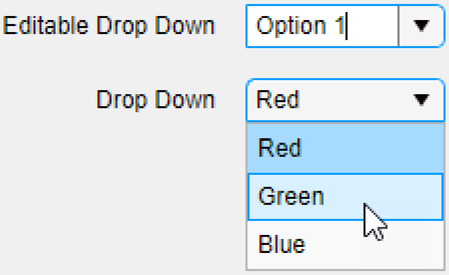
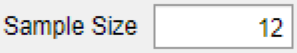
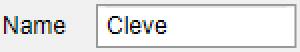
- Common Components — Include components that respond to interactions, such as buttons, sliders, drop-down lists, and trees.
- Axes — Include axes to create plots for data visualization and exploration.
- Containers and Figure Tools — Include panels and tabs for grouping components, as well as menu bars.
- Instrumentation Components — Include gauges and lamps for visualizing status, as well as knobs and switches for selecting input parameters.
- Extensible Components — Include custom UI components that you author. Interface with third-party libraries to display content like widgets or data visualizations.
- Toolbox Components — Include toolbox authored UI components. Requires additional toolbox license and installation.

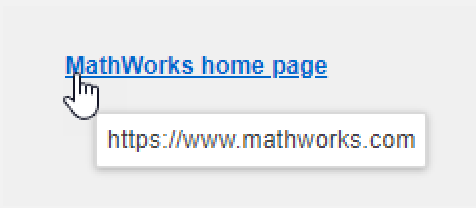
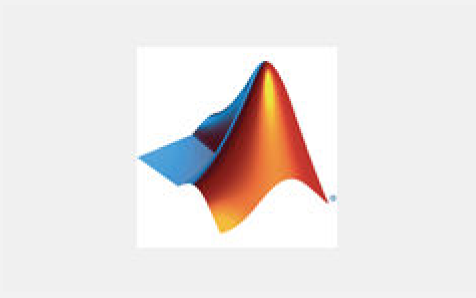
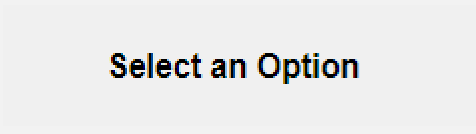
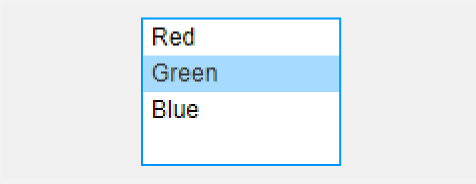
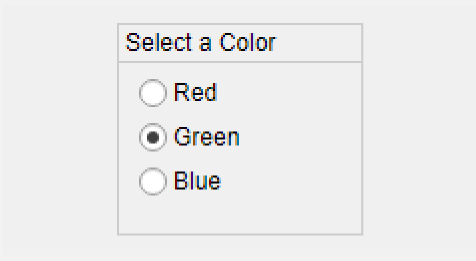
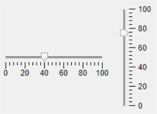
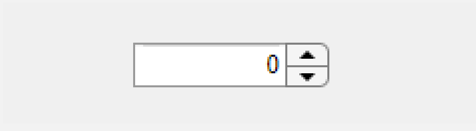
All components are available programmatically. Most UI components are also available in the App Designer **Component Library** for you to drag and drop onto the canvas. To add components to an App Designer app that are not available in the **Component Library**, or that you want to add dynamically to the running app, see “Add UI Components to App Designer Programmatically” on page 4-20.

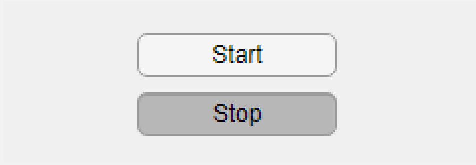
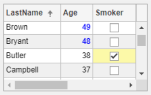
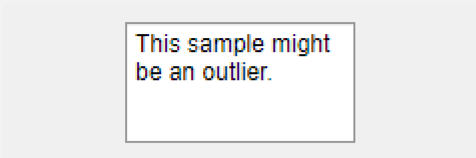
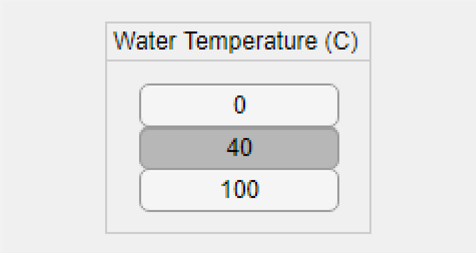
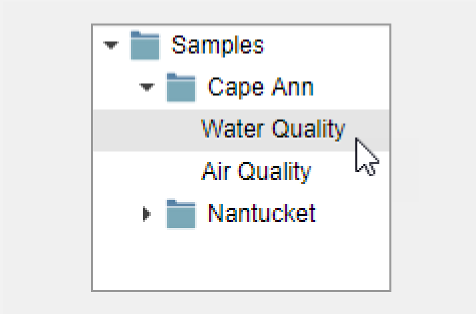
When calling graphics functions in App Designer, the workflow is slightly different than you typically use at the MATLAB command line. For more information about how to call graphics functions in App Designer, see “Display Graphics in App Designer” on page 3-15.

Common Components

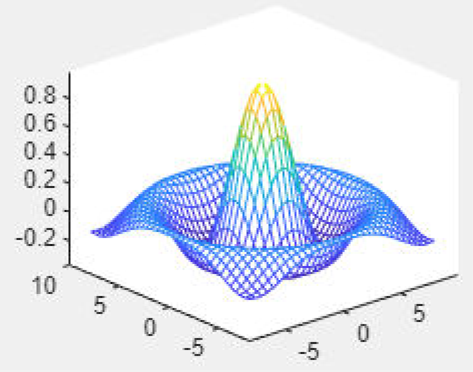
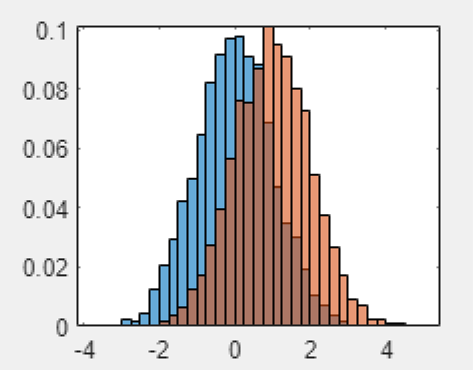
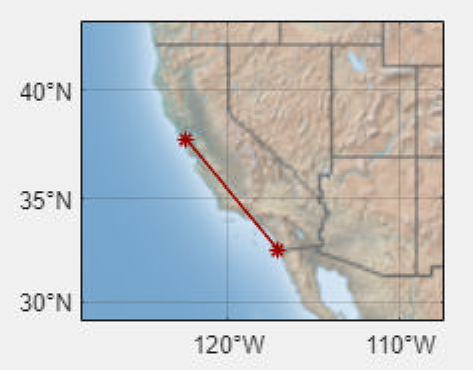
Component Information	Example
Button	
CheckBox	

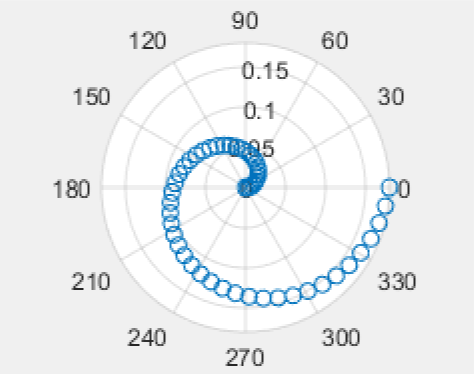
Component Information	Example
CheckBoxTree Properties TreeNode	 □ Sedimentary ☑ Igneous ■ Metamorphic □ Slate ☑ Marble ☑ Gneiss
DatePicker Properties	 mm/dd/yyyy July 2018 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 1 2 3 4 5 6 7 8 9 10 11
DropDown	 Editable Drop Down: Option 1 Drop Down: Red Red Green Blue
NumericEditField	 Sample Size: 12
EditField	 Name: Cleve

Component Information	Example
Hyperlink Properties	 MathWorks home page https://www.mathworks.com
Image Properties	
Label	 Select an Option
ListBox	 Red Green Blue
ButtonGroup RadioButton	 Select a Color ○ Red ● Green ○ Blue
Slider	 0 20 40 60 80 100 100 80 60 40 20 0
Spinner	 0

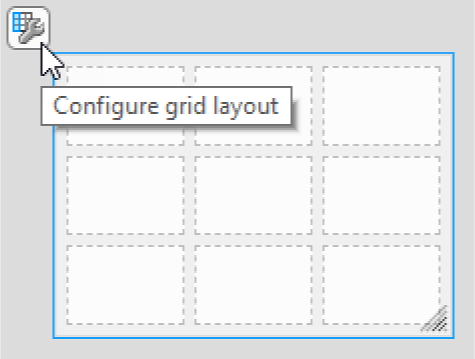
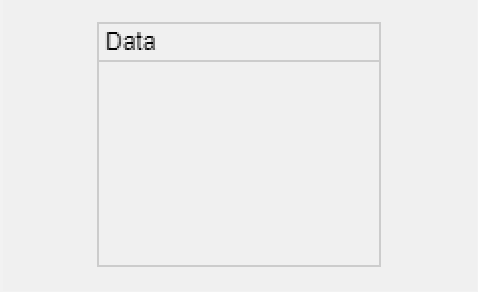
<table>
<tr><th>Component Information</th><th>Example</th></tr>
<tr><td>StateButton</td><td> </td></tr>
<tr><td>Table</td><td>
<table>
<tr><th>LastName ↑</th><th>Age</th><th>Smoker</th></tr>
<tr><td>Brown</td><td>49</td><td>□</td></tr>
<tr><td>Bryant</td><td>48</td><td>□</td></tr>
<tr><td>Butler</td><td>38</td><td>☑</td></tr>
<tr><td>Campbell</td><td>37</td><td>□</td></tr>
</table></td></tr>
<tr><td>TextArea</td><td>
This sample might be an outlier.</td></tr>
<tr><td>ButtonGroup
ToggleButton</td><td> </td></tr>
<tr><td>Tree
TreeNode</td><td> </td></tr>
</table>

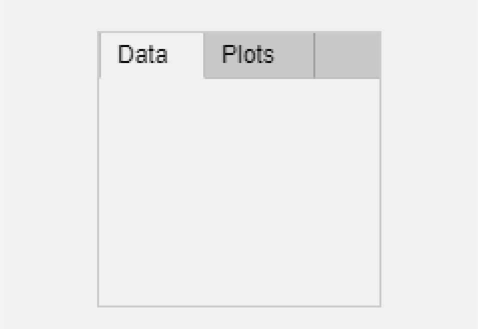
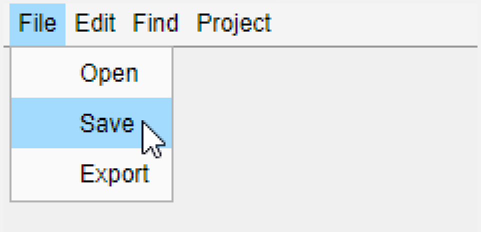
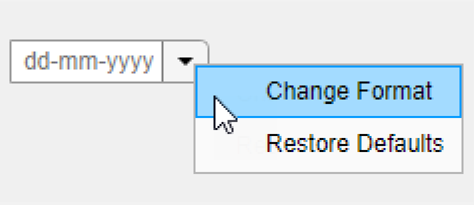
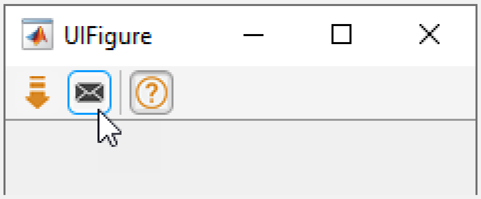
Axes

Axes Information	Example
UIAxes	
Axes Properties This object can be added programmatically only.	
GeographicAxes Properties This object can be added programmatically only.	

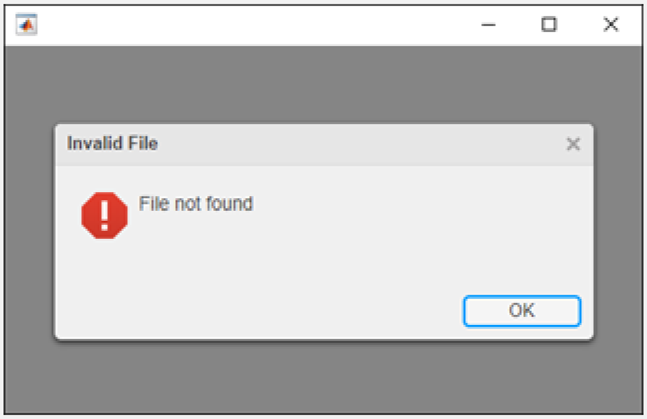
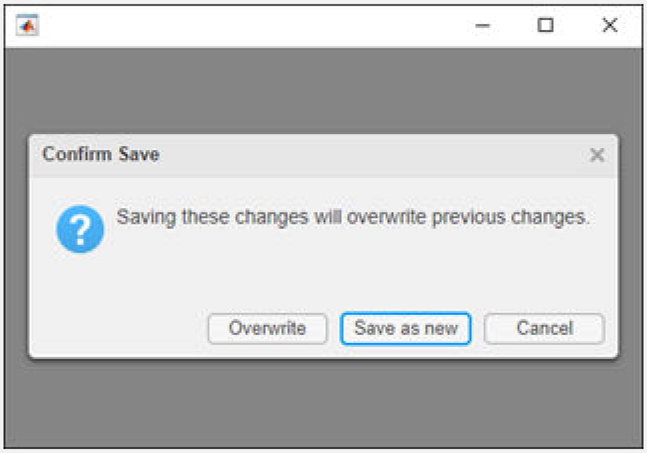
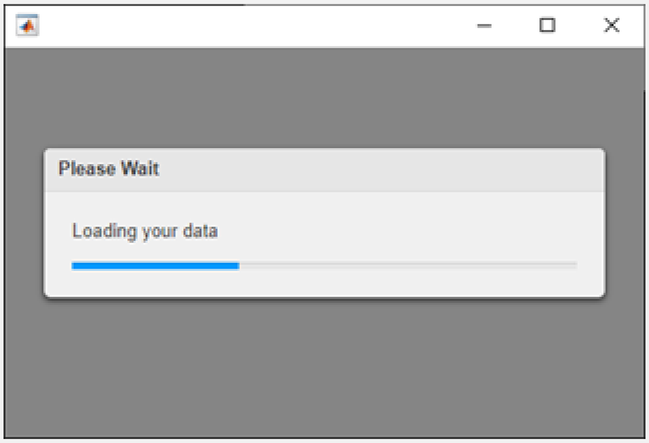
Axes Information	Example
PolarAxes Properties This object can be added programmatically only.	

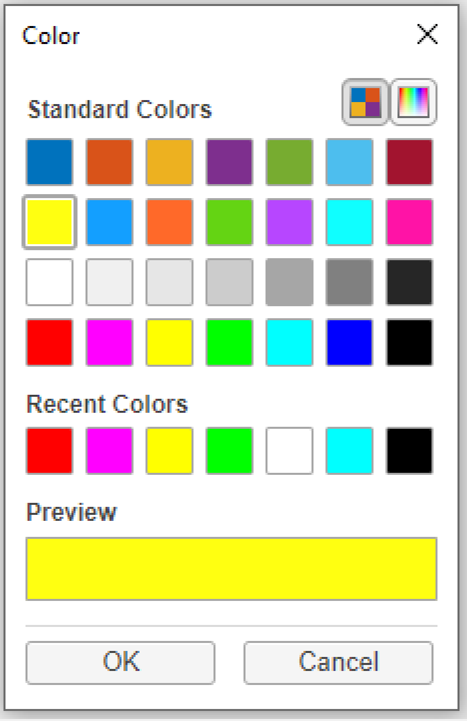
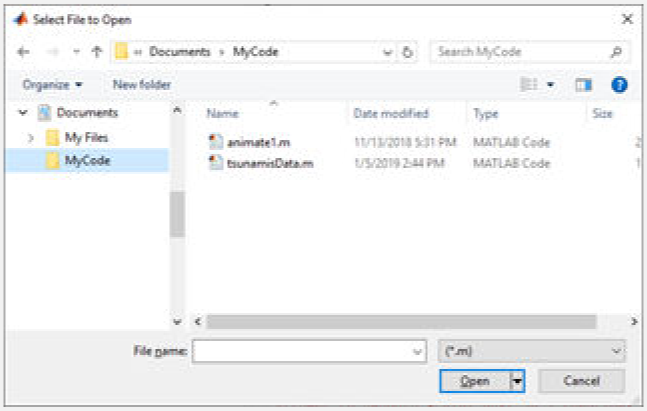
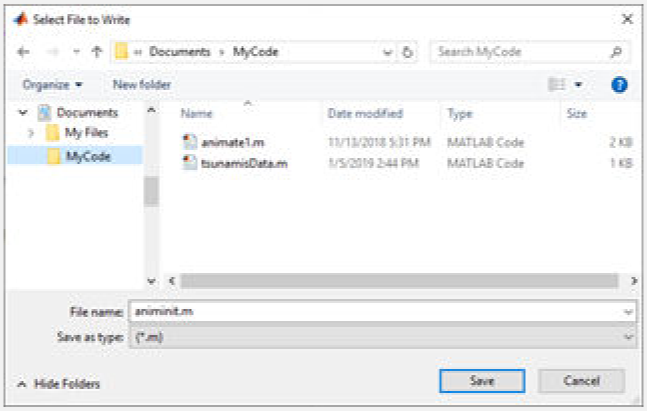
Containers and Figure Tools

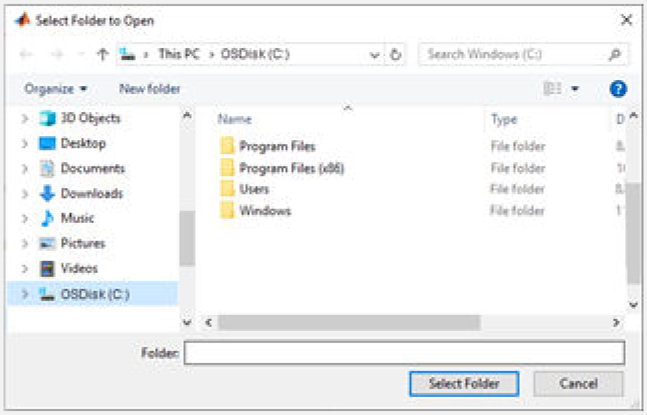
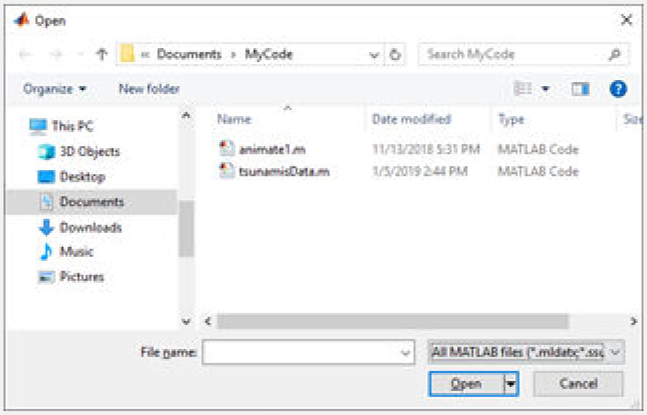
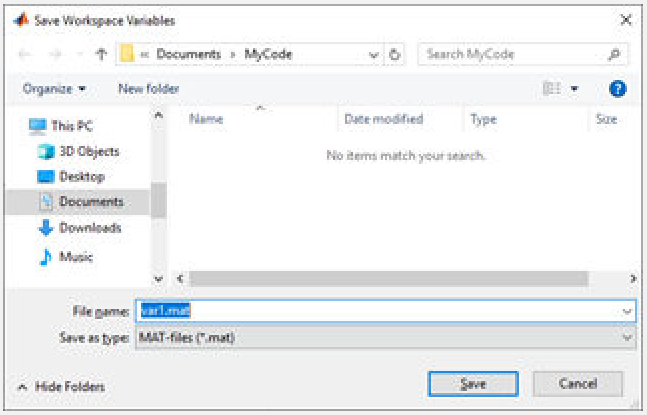
Container Information	Example
GridLayout Properties	
Panel	

Container Information	Example
TabGroup Tab	
Menu	
ContextMenu Properties	
Toolbar Properties PushTool Properties ToggleTool Properties	

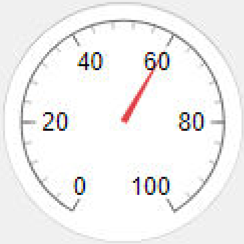
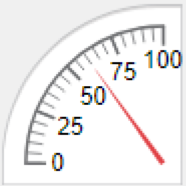
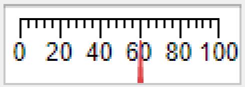
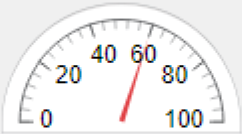
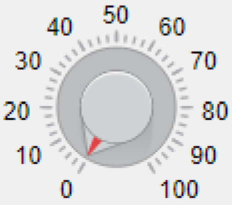
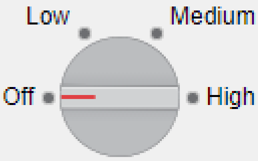
Dialogs and Notifications

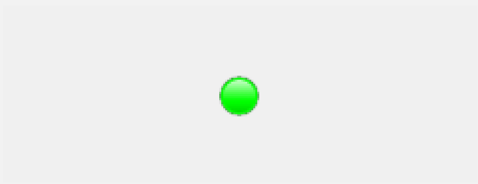
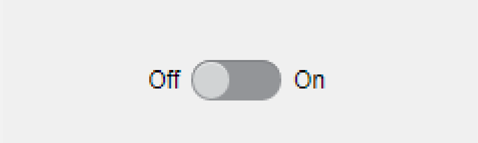
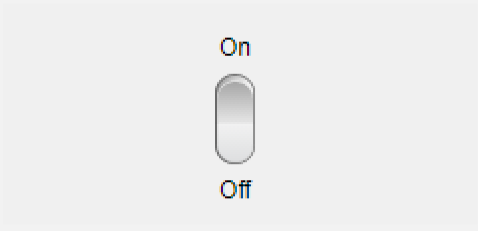
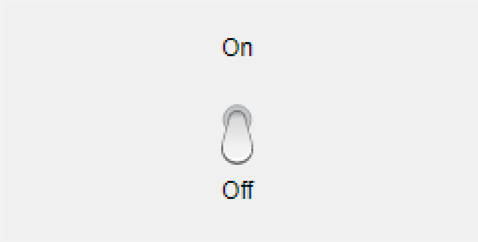
Dialog Information	Example
`uialert` This object can be added programmatically only.	 Invalid File File not found
`uiconfirm` This object can be added programmatically only.	 Confirm Save Saving these changes will overwrite previous changes.
`uiprogressdlg` This object can be added programmatically only.	 Please Wait Loading your data

Dialog Information	Example
`uisetcolor` This object can be added programmatically only.	 Color Standard Colors Recent Colors Preview OK Cancel
`uigetfile` This object can be added programmatically only.	 Select File to Open Documents › MyCode Search MyCode Organize ▾ New folder Documents My Files MyCode Open CancelName · Date modified · Type · Size animate1.m · 11/13/2018 5:31 PM · MATLAB Code tsunamisData.m · 1/5/2019 2:44 PM · MATLAB Code File name: (*.m)
`uiputfile` This object can be added programmatically only.	 Select File to Write Documents › MyCode Search MyCode Organize ▾ New folder Documents My Files MyCode Hide Folders Save CancelName · Date modified · Type · Size animate1.m · 11/13/2018 5:31 PM · MATLAB Code · 2 KB tsunamisData.m · 1/5/2019 2:44 PM · MATLAB Code · 1 KB File name: animinit.m Save as type: (*.m)

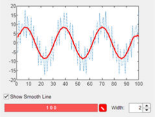
Dialog Information	Example
`uigetdir` This object can be added programmatically only.	 Select Folder to Open This PC › OSDisk (C:) Search Windows (C:) Organize ▾ New folder 3D Objects Desktop Documents Downloads Music Pictures Videos OSDisk (C:) Name Type D Folder: Select Folder Cancel Program Files — File folder Program Files (x86) — File folder Users — File folder Windows — File folder
`uiopen` This object can be added programmatically only.	 Open « Documents › MyCode Search MyCode Organize ▾ New folder This PC 3D Objects Desktop Documents Downloads Music Pictures Name Date modified Type Size File name: Open Cancel animate1.m — 11/13/2018 5:31 PM — MATLAB Code tsunamisData.m — 1/5/2019 2:44 PM — MATLAB Code All MATLAB files (*.mldatx;*.ssc
`uisave` This object can be added programmatically only.	 Save Workspace Variables « Documents › MyCode Search MyCode Organize ▾ New folder This PC 3D Objects Desktop Documents Downloads Music Name Date modified Type Size Hide Folders Save Cancel No items match your search. File name: var1.mat Save as type: MAT-files (*.mat)

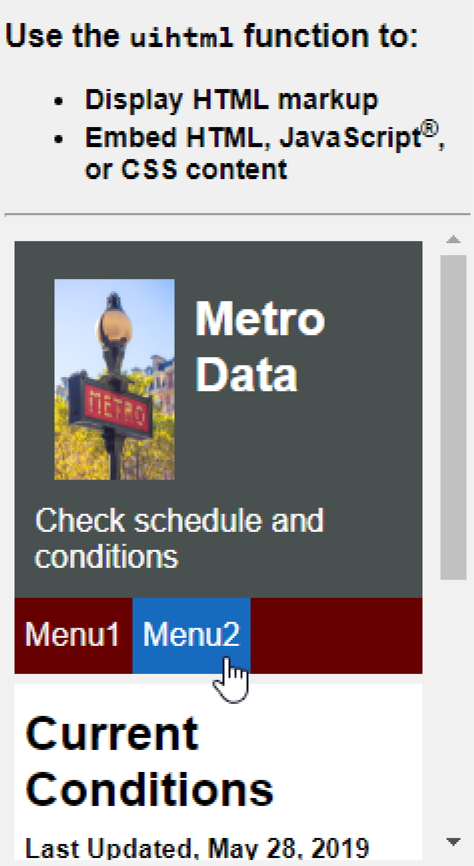
Instrumentation

Component Information	Example
Gauge	
NinetyDegreeGauge	
LinearGauge	
SemicircularGauge	
Knob	 0 10 20 30 40 50 60 70 80 90 100
DiscreteKnob	

Component Information	Example
Lamp	
Switch	
RockerSwitch	
ToggleSwitch	

Extensible Components

Component Information	Example
`matlab.ui.componentcontainer.ComponentContainer` Class `matlab.graphics.chartcontainer.ChartContainer` Class	 20 15 10 5 0 -5 -10 -15 -20 0 10 20 30 40 50 60 70 80 90 100 ☑ Show Smooth Line 100 Width: 2

Component Information	Example
HTML Properties	 Use the `uihtml` function to: • Display HTML markup • Embed HTML, JavaScript®, or CSS content Metro Data Check schedule and conditions Menu1 Menu2 Current Conditions Last Updated, May 28, 2019

Toolbox Components

Apps created in App Designer or with the `uifigure` function support Aerospace Toolbox components. For more information, see “Flight Instruments” (Aerospace Toolbox). To use toolbox components, a valid license and installation of the associated toolbox is required.

See Also

Related Examples

- “Ways to Build Apps” on page 1-2
- “Display Graphics in App Designer” on page 3-15
- “Create and Run a Simple App Using App Designer” on page 3-2
- “Add UI Components to App Designer Programmatically” on page 4-20
- “Create and Run a Simple Programmatic App” on page 15-2

Display Tabular Data in Apps

Table arrays are useful for storing tabular data as MATLAB variables. For example, you can call the `readtable` function to create a table array from a spreadsheet.

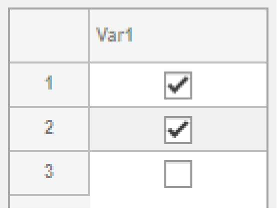
`Table` UI components, by contrast, are user interface components that display tabular data in apps. Starting in R2018a, the types of data you can display in a `Table` UI component include table arrays. Only App Designer apps and figures created with the `uifigure` function support table arrays.

When you display table array data in apps, you can take advantage of the interactive features for certain data types. And unlike other types of arrays that `Table` UI components support, table array data does not display according to the `ColumnFormat` property of the `Table` UI component.

Logical Data

In a `Table` UI component, logical values display as check boxes. `true` values are checked, whereas `false` values are unchecked. When the `ColumnEditable` property of the `Table` UI component is `true`, the user can select and clear the check boxes in the app.

```
fig = uifigure;
tdata = table([true; true; false]);
uit = uitable(fig,'Data',tdata);
uit.Position(3) = 130;
uit.RowName = 'numbered';
```

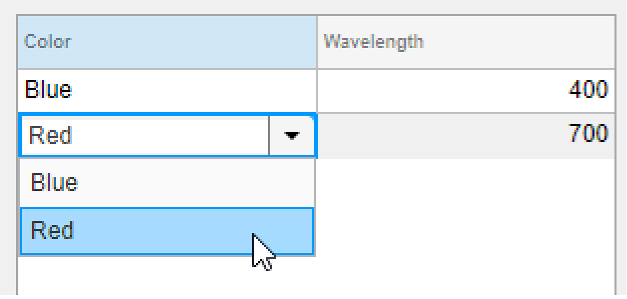


	Var1
1	☑
2	☑
3	□

Categorical Data

`categorical` values can appear as drop-down lists or as text. The categories appear in drop-down lists when the `ColumnEditable` property of the `Table` UI component is `true`. Otherwise, the categories display as text without a drop-down list.

```
fig = uifigure;
cnames = categorical({'Blue';'Red'},{'Blue','Red'});
w = [400; 700];
tdata = table(cnames,w,'VariableNames',{'Color','Wavelength'});
uit = uitable(fig,'Data',tdata,'ColumnEditable',true);
```



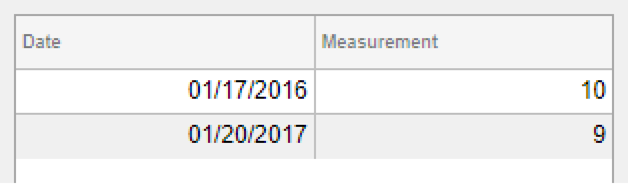
Color	Wavelength
Blue	400
Red	700

If the categorical array is not protected, users can add new categories in the running app by typing in the cell.

Datetime Data

datetime values display according to the Format property of the corresponding table variable (a datetime array).

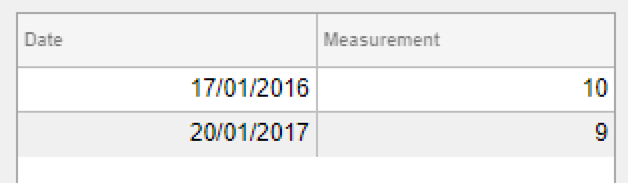
```
fig = uifigure;
dates = datetime([2016,01,17; 2017,01,20],'Format','MM/dd/uuuu');
m = [10; 9];
tdata = table(dates,m,'VariableNames',{'Date','Measurement'});
uit = uitable(fig,'Data',tdata);
```



Date	Measurement
01/17/2016	10
01/20/2017	9

To change the format, use dot notation to set the Format property of the table variable. Then, replace the data in the Table UI component.

```
tdata.Date.Format = 'dd/MM/uuuu';
uit.Data = tdata;
```



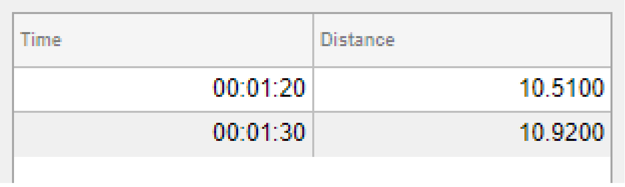
Date	Measurement
17/01/2016	10
20/01/2017	9

When the ColumnEditable property of the Table UI component is true, users can change date values in the app. When the column is editable, the app expects input values that conform to the Format property of the datetime array. If the user enters an invalid date, the value displayed in the table is NaT.

Duration Data

duration values display according to the Format property of the corresponding table variable (a duration array).

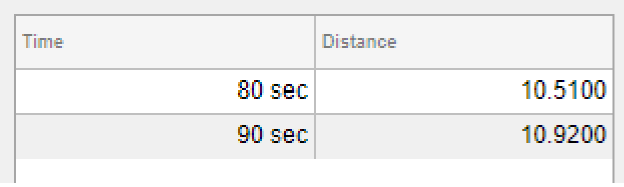
```
fig = uifigure;
mtime = duration([0;0],[1;1],[20;30]);
dist = [10.51; 10.92];
tdata = table(mtime,dist,'VariableNames',{'Time','Distance'});
uit = uitable(fig,'Data',tdata);
```



Time	Distance
00:01:20	10.5100
00:01:30	10.9200

To change the format, use dot notation to set the `Format` property of the table variable.

```
tdata.Time.Format = 's';
uit.Data = tdata;
```



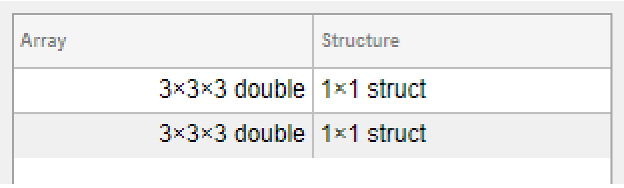
Time	Distance
80 sec	10.5100
90 sec	10.9200

Cells containing `duration` values are not editable in the running app, even when `ColumnEditable` of the `Table` UI component is `true`.

Nonscalar Data

Nonscalar values display in the app the same way as they display in the Command Window. For example, this table array contains 3-D arrays and `struct` arrays.

```
fig = uifigure;
arr = {rand(3,3,3); rand(3,3,3)};
s = {struct; struct};
tdata = table(arr,s,'VariableNames',{'Array','Structure'});
uit = uitable(fig,'Data',tdata);
```



Array	Structure
3×3×3 double	1×1 struct
3×3×3 double	1×1 struct

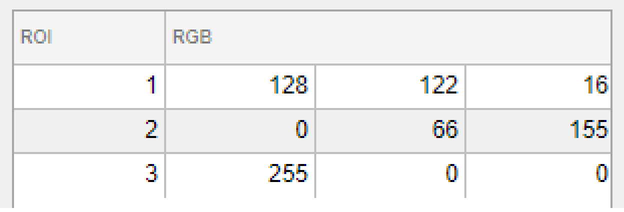
A multicolumn table array variable displays as a combined column in the app, just as it does in the Command Window. For example, the RGB variable in this table array is a 3-by-3 array.

```
n = [1;2;3];
rgbs = [128 122 16; 0 66 155; 255 0 0];
tdata = table(n,rgbs,'VariableNames',{'ROI','RGB'})
```

```
tdata = 

  3×2 table

    ROI           RGB        
    ___    _________________

     1     128    122     16
     2       0     66    155
     3     255      0      0
```

The `Table` UI component provides a similar presentation. Selecting an item in the RGB column selects all the subcolumns in that row. The values in the subcolumns are not editable in the running app, even when `ColumnEditable` property of the `Table` UI component is `true`.

```
fig = uifigure;
uit = uitable(fig,'Data',tdata);
```



ROI	RGB		
1	128	122	16
2	0	66	155
3	255	0	0

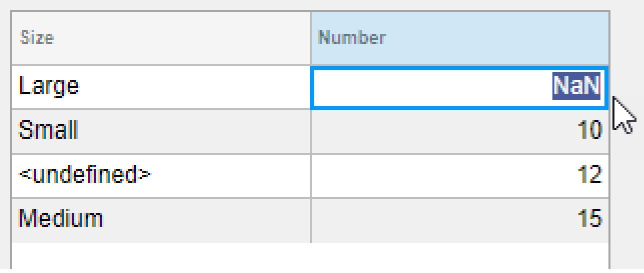
Missing Data Values

Missing values display as indicators according to the data type:

- Missing strings display as `<missing>`.
- Undefined `categorical` values display as `<undefined>`.
- Invalid or undefined numbers or `duration` values display as `NaN`.
- Invalid or undefined `datetime` values display as `NaT`.

If the `ColumnEditable` property of the `Table` UI component is `true`, then the user can correct the values in the running app.

```
fig = uifigure;
sz = categorical([1; 3; 4; 2],1:3,{'Large','Medium','Small'});
num = [NaN; 10; 12; 15];
tdata = table(sz,num,'VariableNames',{'Size','Number'});
uit = uitable(fig,'Data',tdata,'ColumnEditable',true);
```

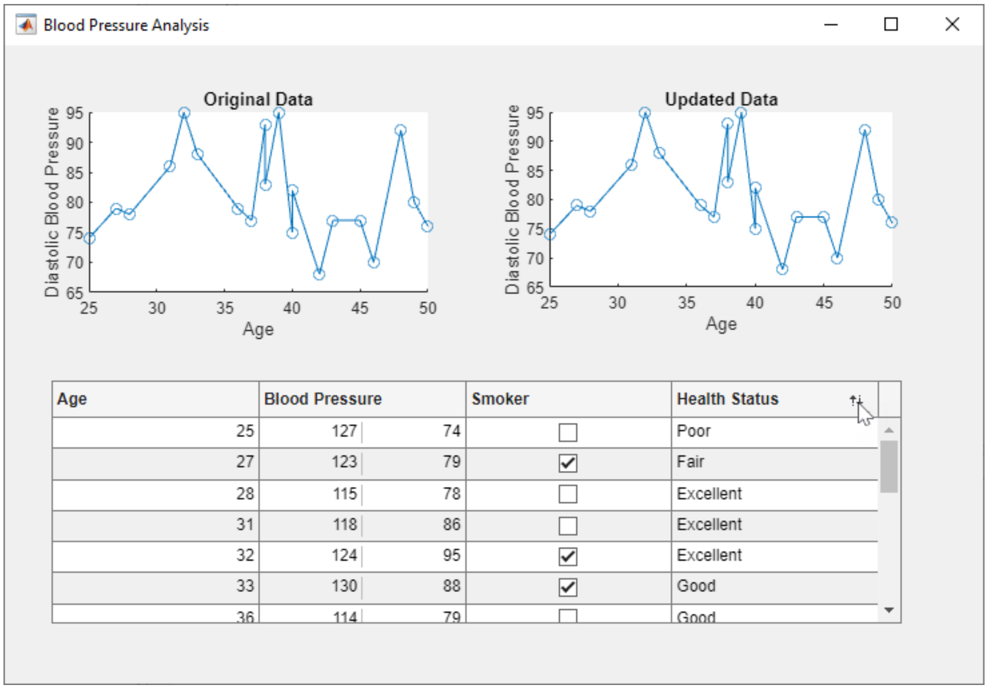


Size	Number
Large	NaN
Small	10
<undefined>	12
Medium	15

Example: App That Displays a Table

This example shows how to display a table UI component in an app that uses table data. The table contains `numeric`, `logical`, `categorical`, and multicolumn variables.

The `StartupFcn` callback loads a spreadsheet into a table array. Then a subset of the data displays and is plotted in the app. One plot displays the original table data. The other plot initially shows the same table data and then updates when the user edits a value or sorts a column in the table UI component.



Age	Blood Pressure		Smoker	Health Status
25	127	74	□	Poor
27	123	79	☑	Fair
28	115	78	□	Excellent
31	118	86	□	Excellent
32	124	95	☑	Excellent
33	130	88	☑	Good
36	114	79	□	Good

See Also

Table (App Designer) | `uitable`

Related Examples

- “Callbacks in App Designer” on page 6-16
- “Reuse Code Using Helper Functions” on page 6-23

Add UI Components to App Designer Programmatically

Most UI components are available in the App Designer **Component Library** for you to drag and drop onto the canvas. Occasionally, you might need to add components programmatically in Code View. Here are a few common situations:

- Creating components that are not available in the **Component Library**. For example, an app that displays a dialog box must call the appropriate function to display the dialog box.
- Creating components dynamically according to run-time conditions.

When you add UI components programmatically, you must call the appropriate function to create the component, assign a callback to the component, and then write the callback as a helper function.

Create the Component and Assign the Callback

Call the function that creates the component from within an existing callback (for a list of UI component functions, see “Develop uifigure-Based Apps”). The `StartupFcn` callback is a good place to create components because that callback runs when the app starts up. In other cases, you might create components within a different callback function. For example, if you want to display a dialog box when the user presses a button, call the dialog box function from within the button's callback function.

When you call a function to create a component, specify the figure or one of its child containers as the parent object. For example, this command creates a button and specifies the figure as the parent object. In this case, the figure has the default name that App Designer assigns (`app.UIFigure`).

```
b = uibutton(app.UIFigure);
```

Next, specify the component's callback property as a function handle of the form `@app.`*`callbackname`*. For example, this command sets the `ButtonPushedFcn` property of button `b` to a callback function named `mybuttonpress`.

```
b.ButtonPushedFcn = @app.mybuttonpress;
```

Write the Callback

Write the callback function for the component as a private helper function. The function must have `app`, `src`, and `event` as the first three arguments. Here is an example of a callback written as a private helper function.

```
methods (Access = private)

        function mybuttonpress(app,src,event)
            disp('Have a nice day!');
        end

end
```

To write a callback that accepts additional input arguments, specify the additional arguments after the first three. For example, this callback accepts two additional inputs, `x` and `y`:

```
methods (Access = private)

        function addxy(app,src,event,x,y)
```

```
                disp(x + y);
            end

    end
```

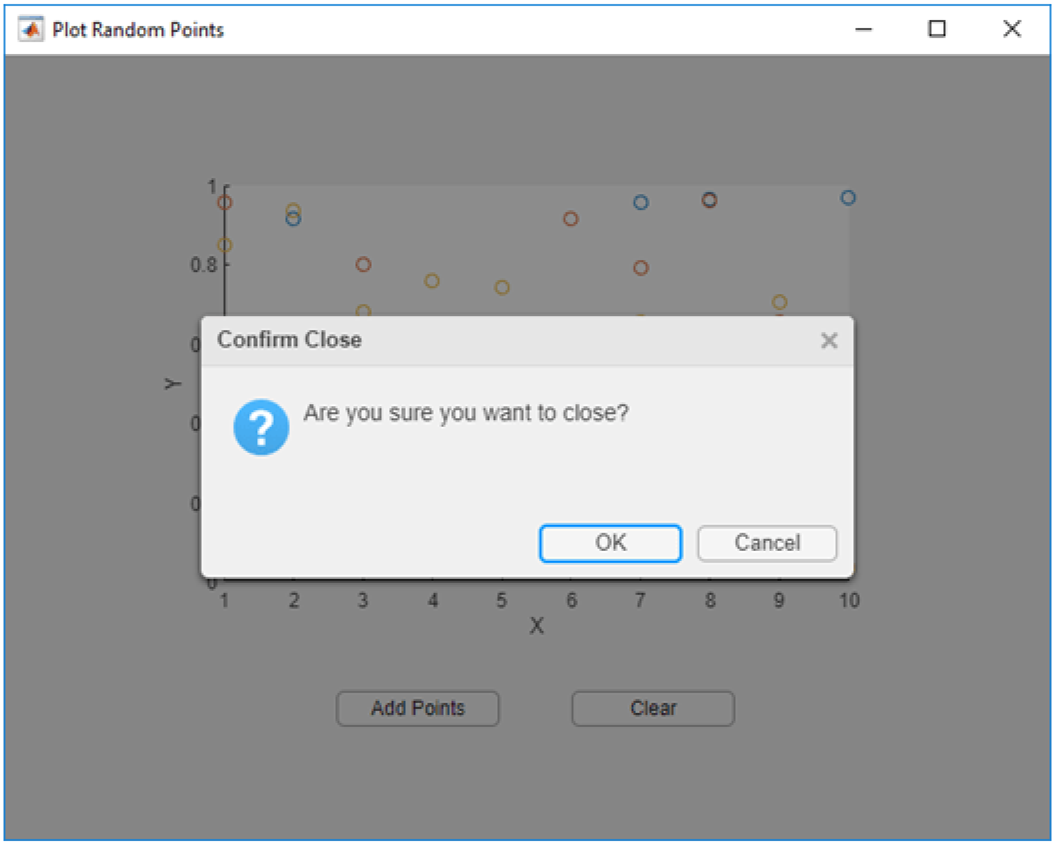
To assign this callback to a component, specify the component's callback property as cell array. The first element in the cell array must be the function handle. Subsequent elements must be the additional input values. For example:

```
b.ButtonPushedFcn = {@app.addxy,10,20};
```

Example: Confirmation Dialog Box with a Close Function

This app shows how to display a confirmation dialog box that executes a callback when the dialog box closes.

When the user clicks the window's close button (**X**), a dialog box displays to confirm that the user wants to close the app. When the user dismisses the dialog box, the `CloseFcn` callback executes.

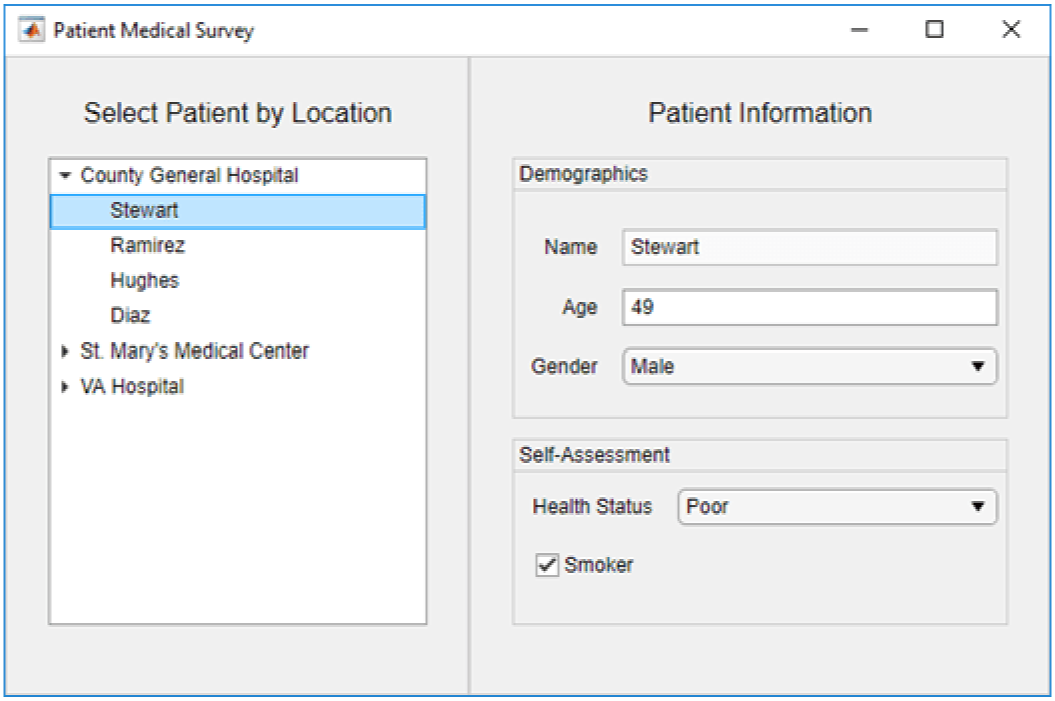


Are you sure you want to close?

Example: App that Populates Tree Nodes Based on a Data File

This app shows how to dynamically add tree nodes at run time. The three hospital nodes exist in the tree before the app runs. However at run time, the app adds several child nodes under each hospital name. The number of child nodes and the labels on the child nodes are determined by the contents of the `patients.xls` spreadsheet.

When the user clicks a patient name in the tree, the **Patient Information** panel displays data such as age, gender, and health status. The app stores changes to the data in a table array.



Name Stewart

Age 49

Gender Male

Health Status Poor

☑ Smoker

See Also

More About

- “Callbacks in App Designer” on page 6-16
- “Reuse Code Using Helper Functions” on page 6-23

Create HTML Content in Apps

You can add HTML content, including JavaScript, CSS, and third-party visualizations or widgets, to your app by using an HTML UI component. Use the `uihtml` function to create an HTML UI component.

When you add an HTML UI component to your app, write code to communicate between MATLAB and JavaScript. You can set component data, respond to changes in data, and react to user interaction by sending events.

Communicate Between MATLAB and JavaScript

To connect the MATLAB HTML UI component in your app to your HTML content, implement a `setup` function in your HTML file. The `setup` function defines and initializes a local JavaScript `htmlComponent` object, which synchronizes with the MATLAB `HTML` object. The JavaScript `htmlComponent` object is accessible only from within the `setup` function.

The `setup` function executes whenever one of these events happens:

- The HTML UI component is created in the UI figure and the content has fully loaded.
- The `HTMLSource` property of the MATLAB `HTML` object changes to a new value.

With this connection, you can share information between your MATLAB and JavaScript code using multiple approaches:

- Share component data — Use this approach when your HTML component has static data that you need to access from both your MATLAB and JavaScript code. For example, if your component contains a table, store the table data as shared component data.
- Send component events — Use this approach to broadcast a notification of a change or interaction. You can send an event from JavaScript to MATLAB or from MATLAB to JavaScript. For example, send an event from JavaScript when a user clicks a button HTML element to react to this interaction in your MATLAB code.

This table gives an overview of the ways that the MATLAB `HTML` object and the JavaScript `htmlComponent` object can communicate.

Task	MATLAB	JavaScript

Access component object.	MATLAB represents the UI component as an `HTML` object.
	You can access the `HTML` object from MATLAB by storing the output of the `uihtml` function as a variable.
	`fig = uifigure;`
	`c = uihtml(fig);`
		JavaScript represents the UI component as the `htmlComponent` object.
		You can access the `htmlComponent` object only from within the `setup` function of the HTML source file associated with the MATLAB `HTML` object.
		`<script type="text/javascript">`
		`    function setup(htmlComponent) {`
		`        // Access the htmlComponent object h`
		`    }`
		`</script>`

Task	MATLAB	JavaScript

Access component data.	The MATLAB `HTML` object has a `Data` property. This property is synchronized with the `Data` property of the JavaScript `htmlComponent` object. Use this property to transfer data to or access data from your HTML source code.
	Access the property in your MATLAB code.
	`fig = uifigure;`
	`c = uihtml(fig);`
	`c.Data = 10;`
		The JavaScript `htmlComponent` object has a `Data` property. This property is synchronized with the `Data` property of the MATLAB `HTML` object. Use this property to transfer data to or access data from MATLAB.
		Access the property in the `setup` function of the HTML source file.
		`<script type="text/javascript">`
		`    function setup(htmlComponent) {`
		`        htmlComponent.Data = 5;`
		`    }`
		`</script>`

Respond to a change in component data.	The MATLAB `HTML` object has a `DataChangedFcn` callback property.
	Create a `DataChangedFcn` callback for the `HTML` UI component to execute a function when the `Data` property of the `htmlComponent` JavaScript object changes.
	`fig = uifigure;`
	`c = uihtml(fig);`
	`c.DataChangedFcn = @(src,event) disp(event.Data);`
		The JavaScript `htmlComponent` object has an `addEventListener` method.
		Add a `DataChanged` listener to the `htmlComponent` object to execute a function when the `Data` property of the MATLAB `HTML` object changes.
		`htmlComponent.addEventListener("DataChanged"`
		`function updateData(event) {`
		`    let changedData = htmlComponent.Data;`
		`    // Update HTML or JavaScript with the ne`
		`}`
		For more information about the `addEventListener` method, see EventTarget.addEventListener() on Mozilla® MDN web docs.

Send and react to an event from MATLAB in JavaScript.	To send an event from MATLAB to JavaScript, use the `sendEventToHTMLSource` function.
	For example, you can send an event to react to a user interacting with a MATLAB UI component in your JavaScript code.
	`fig = uifigure;`
	`c = uihtml(fig);`
	`eventData = [1 2 3];`
	`sendEventToHTMLSource(c,"myMATLABEvent",eventData);`
		To react to an event from MATLAB in your JavaScript code, add a listener that listens for the MATLAB event to the JavaScript `htmlComponent` object.
		Access event data sent from MATLAB using the `event.Data` property.
		`htmlComponent.addEventListener("myMATLABEven`
		`function processEvent(event) {`
		`    let eventData = event.Data;`
		`    // Update HTML or JavaScript to react to`
		`}`

Task	MATLAB	JavaScript

Send and react to an event from JavaScript in MATLAB.	To react to an event from JavaScript in your MATLAB code, create an `HTMLEventReceivedFcn` callback for your `HTML` UI component.
	`fig = uifigure;`
	`c = uihtml(fig);`
	`c.HTMLEventReceivedFcn = @(src,event) disp(event);`
		To send an event from JavaScript to MATLAB, use the `sendEventToMATLAB` function.
		For example, you can send an event to react to a user clicking an HTML button element in your MATLAB code.
		`eventData = [1,2,3];`
		`htmlComponent.sendEventToMATLAB("myHTMLEvent`

For an example of an HTML source file that is configured to connect to a MATLAB `HTML` UI component, see “Sample HTML Source File” on page 4-25.

Convert Data Between MATLAB and JavaScript

You can pass two types of data between the MATLAB HTML component and the JavaScript `htmlComponent` object:

- Component data, stored in the `Data` property of each object
- Event data, associated with an event sent from MATLAB to JavaScript or JavaScript to MATLAB

Because MATLAB and JavaScript support slightly different sets of data types, the component converts the data when it is shared.

When the component converts data from MATLAB to JavaScript:

1. The component encodes the MATLAB data as JSON-formatted text using the `jsonencode` function.
2. The component parses the JSON-formatted text to JavaScript data using JSON.parse().

When the component converts data from JavaScript to MATLAB:

1. The component encodes the JavaScript data as JSON-formatted text using JSON.stringify().
2. The component parses the JSON-formatted text to MATLAB data using the `jsondecode` function.

You can use these functions to simulate how your data is sent between MATLAB and JavaScript to help you write and debug your code. For more information, see “Debug HTML Content in Apps” on page 4-30.

Sample HTML Source File

This example provides a sample HTML source file. Save this code to a file named `sampleHTMLFile.html`. You can use this sample file as a starting point for your own HTML UI components, or to explore how a component sends data between MATLAB and JavaScript.

The sample file creates three elements:

- An edit field to display and edit component data
- An edit field to display and edit event data

- A button to send an event from JavaScript to MATLAB

The `setup` function in the sample file defines four callback functions:

- `dataFromMATLABToHTML` — Update the **Component data** edit field with the current data. This function executes whenever the `Data` property of the MATLAB `HTML` object changes.
- `eventFromMATLABToHTML` — Update the **Event data** edit field with the data from the most recent event. This function executes whenever MATLAB sends an event named `"MyMATLABEvent"` to the HTML source.
- `dataFromHTMLToMATLAB` — Update the `Data` property of the JavaScript `htmlComponent` object with the text in the **Component data** edit field. This function executes whenever a user enters a new value in the edit field. The function triggers the `DataChangedFcn` callback of the MATLAB `HTML` object.
- `eventFromHTMLToMATLAB` — Send an event named `"MyHTMLSourceEvent"` with data from the text in the **Event data** edit field. This function executes whenever a user clicks the **Send event to MATLAB** button. The function triggers the `HTMLEventReceivedFcn` callback of the MATLAB `HTML` object.

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript">

        function setup(htmlComponent) {
            console.log("Setup called:", htmlComponent);

            // Code response to data changes in MATLAB
            htmlComponent.addEventListener("DataChanged", dataFromMATLABToHTML);

            // Code response to events from MATLAB
            htmlComponent.addEventListener("MyMATLABEvent", eventFromMATLABToHTML);

            // Update the Data property of the htmlComponent object
            // This action also updates the Data property of the MATLAB HTML object
            // and triggers the DataChangedFcn callback function
            let dataInput = document.getElementById("compdata")
            dataInput.addEventListener("change", dataFromHTMLToMATLAB);

            // Send an event to MATLAB and trigger
            // the HTMLEventReceivedFcn callback function
            let eventButton = document.getElementById("send");
            eventButton.addEventListener("click", eventFromHTMLToMATLAB);

            function dataFromMATLABToHTML(event) {
                let changedData = htmlComponent.Data;
                console.log("New data from MATLAB:", changedData);

                // Update your HTML or JavaScript with the new data
                let dom = document.getElementById("compdata");
                dom.value = changedData;
            }

            function eventFromMATLABToHTML(event) {
                let eventData = event.Data;
                console.log("Event from MATLAB with event data:", eventData);
                // Update your HTML or JavaScript to react to the event
                let dom = document.getElementById("evtdata");
                dom.value = eventData;
            }

            function dataFromHTMLToMATLAB(event) {
                newData = event.target.value;
                htmlComponent.Data = newData;
                console.log("New data in HTML:", newData)
            }

            function eventFromHTMLToMATLAB(event) {
                eventData = document.getElementById("evtdata").value;
                htmlComponent.sendEventToMATLAB("MyHTMLSourceEvent", eventData);
                console.log("Sending event to MATLAB with event data:", eventData);
            }
        }
    </script>
</head>

<body>
    <div style="font-family:sans-serif;">
        <label for="compdata">Component data:</label>
        <input type="text" id="compdata" name="compdata"><br><br>
        <label for="evtdata">Event data:</label>
        <input type="text" id="evtdata" name="evtdata"><br><br>
        <button id="send">Send event to MATLAB</button>
    </div>
</body>

</html>
```

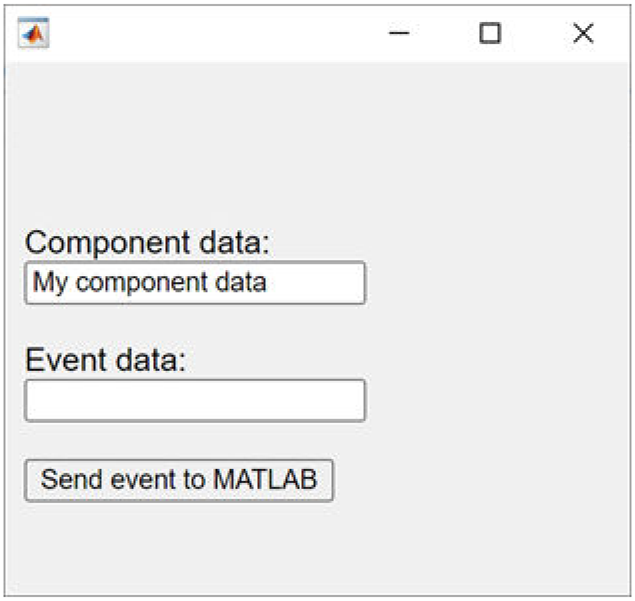
Send Data and Events Between MATLAB and JavaScript

In MATLAB, create an HTML UI component and specify the HTML source as `sampleHTMLFile.html`. Assign `DataChangedFcn` and `HTMLEventReceivedFcn` callbacks that display the component data and event data, respectively.

```
fig = uifigure;
h = uihtml(fig, ...
    "HTMLSource","sampleHTMLFile.html", ...
    "DataChangedFcn",@(src,event) disp(src.Data), ...
    "HTMLEventReceivedFcn",@(src,event) disp(event.HTMLEventData), ...
    "Position",[20 20 200 200]);
```

Specify the component `Data` property in MATLAB. The **Component data** field updates to display the data.

```
h.Data = "My component data";
```

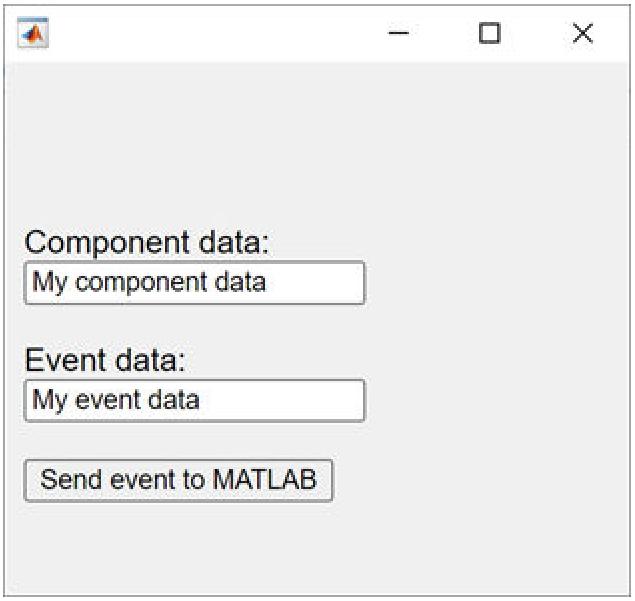


Component data:
My component data

Event data:

Send an event from MATLAB to JavaScript and specify some event data. The **Event data** field updates to display the data.

```
sendEventToHTMLSource(h,"MyMATLABEvent","My event data")
```



Component data:
My component data

Event data:
My event data

Update the text in the **Component data** field, then press **Enter**. In MATLAB, the `DataChangedFcn` callback executes and displays the updated text in the Command Window.

Finally, update the text in the **Event data** field, then click the **Send event to MATLAB** button. In MATLAB, the `HTMLEventReceivedFcn` callback executes and displays the updated text in the Command Window.

See Also

Functions
`uihtml` | `sendEventToHTMLSource` | `jsonencode` | `jsondecode`

Properties
HTML Properties

See Also

Related Examples

- “Debug HTML Content in Apps” on page 4-30
- “Display HTML Elements Styled by a Cascading Style Sheet” on page 7-15

Debug HTML Content in Apps

You can include HTML content, including JavaScript, CSS, and third-party visualizations or widgets, in your app by using an HTML UI component. If you create an app with an HTML UI component that is not working as expected, or if you want to know what your data looks like when converting between MATLAB and JavaScript, you can use your system browser to debug the code in the HTML source file. Using the Developer Tools (DevTools) of your browser, you can set breakpoints to test portions of your `setup` function. When you debug your HTML file through the system browser, you must simulate the connection between MATLAB and JavaScript that the `setup` function provides.

Create Sample HTML Source File

Create a sample HTML source file. Save this code to a file named `sampleHTMLFile.html`.

The sample file creates three elements:

- An edit field to display and edit component data
- An edit field to display and edit event data
- A button to send an event from JavaScript to MATLAB

The `setup` function in the sample file defines four callback functions:

- `dataFromMATLABToHTML` — Update the **Component data** edit field with the current data. This function executes whenever the `Data` property of the MATLAB `HTML` object changes.
- `eventFromMATLABToHTML` — Update the **Event data** edit field with the data from the most recent event. This function executes whenever MATLAB sends an event named `"MyMATLABEvent"` to the HTML source.
- `dataFromHTMLToMATLAB` — Update the `Data` property of the JavaScript `htmlComponent` object with the text in the **Component data** edit field. This function executes whenever a user enters a new value in the edit field. The function triggers the `DataChangedFcn` callback of the MATLAB `HTML` object.
- `eventFromHTMLToMATLAB` — Send an event named `"MyHTMLSourceEvent"` with data from the text in the **Event data** edit field. This function executes whenever a user clicks the **Send event to MATLAB** button. The function triggers the `HTMLEventReceivedFcn` callback of the MATLAB `HTML` object.

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript">

        function setup(htmlComponent) {
            console.log("Setup called:", htmlComponent);

            // Code response to data changes in MATLAB
            htmlComponent.addEventListener("DataChanged", dataFromMATLABToHTML);

            // Code response to events from MATLAB
            htmlComponent.addEventListener("MyMATLABEvent", eventFromMATLABToHTML);

            // Update the Data property of the htmlComponent object
            // This action also updates the Data property of the MATLAB HTML object
            // and triggers the DataChangedFcn callback function
            let dataInput = document.getElementById("compdata")
            dataInput.addEventListener("change", dataFromHTMLToMATLAB);

            // Send an event to MATLAB and trigger
            // the HTMLEventReceivedFcn callback function
            let eventButton = document.getElementById("send");
            eventButton.addEventListener("click", eventFromHTMLToMATLAB);

            function dataFromMATLABToHTML(event) {
                let changedData = htmlComponent.Data;
                console.log("New data from MATLAB:", changedData);

                // Update your HTML or JavaScript with the new data
                let dom = document.getElementById("compdata");
                dom.value = changedData;
            }

            function eventFromMATLABToHTML(event) {
                let eventData = event.Data;
                console.log("Event from MATLAB with event data:", eventData);

                // Update your HTML or JavaScript to react to the event
                let dom = document.getElementById("evtdata");
                dom.value = eventData;
            }

            function dataFromHTMLToMATLAB(event) {
                newData = event.target.value;
                htmlComponent.Data = newData;
                console.log("New data in HTML:", newData)
            }

            function eventFromHTMLToMATLAB(event) {
                eventData = document.getElementById("evtdata").value;
                htmlComponent.sendEventToMATLAB("MyHTMLSourceEvent", eventData);
                console.log("Sending event to MATLAB with event data:", eventData);
            }
        }
    </script>
</head>

<body>
    <div style="font-family:sans-serif;">
        <label for="compdata">Component data:</label>
        <input type="text" id="compdata" name="compdata"><br><br>
        <label for="evtdata">Event data:</label>
        <input type="text" id="evtdata" name="evtdata"><br><br>
        <button id="send">Send event to MATLAB</button>
    </div>
</body>

</html>
```

View HTML File in Browser

To debug the code in your HTML source file, first open the file in your browser. From the Current Folder browser in MATLAB, right-click the `sampleHTMLFile.html` file and select **Open Outside MATLAB**.

Once the file is open in your browser, you can use the Developer Tools (DevTools) of your browser to set breakpoints in the file and access the console.

Simulate Sending Data from MATLAB to JavaScript

To debug how the code in your HTML file responds to component data changes from MATLAB, simulate the way that MATLAB sends component data to JavaScript.

1. In MATLAB, run this code to convert a MATLAB cell array of character vectors to a JSON string. Copy the returned string value to your clipboard.

   ```
   value = {'one';'two';'three'};
   jsontxt = jsonencode(value)
   ```

   ```
   jsontxt = 
   '["one","two","three"]'
   ```
2. In the DevTools of your system browser, open the `sampleHTMLFile.html` file to view the code. Create a breakpoint in the `setup` function, after the code creates the `htmlComponent` object and adds the `"DataChanged"` listener.
3. Open the DevTools console and manually create the JavaScript `htmlComponent` object. Use the `JSON.parse` method to convert the JSON string you just generated in MATLAB to a JavaScript object and store it in the `Data` property of the `htmlComponent` object.

   ```
   var htmlComponent = {
       Data: JSON.parse('["one","two","three"]'), // JSON-formatted text from MATLAB data
       addEventListener: function() {console.log("addEventListener called with: ", arguments)},
       sendEventToMATLAB: function() {console.log("sendEventToMATLAB called with: ", arguments)}
   };
   ```
4. While still in the DevTools console, call the `setup` function and pass it the `htmlComponent` object you created.

   ```
   setup(htmlComponent)
   ```
5. While paused at the breakpoint, execute the `dataFromMATLABToHTML` callback function to simulate a component data change in MATLAB. The **Component data** edit field updates in the HTML page.

   ```
   dataFromMATLABToHTML()
   ```

Simulate Sending Events from MATLAB to JavaScript

To debug how the code in your HTML file reacts to an event from MATLAB, simulate the way that MATLAB sends event data to JavaScript.

1. In MATLAB, run this code to convert a MATLAB cell array of character vectors to a JSON string. Copy the returned string value to your clipboard.

   ```
   value = {'one';'two';'three'};
   jsontxt = jsonencode(value)
   ```

```
jsontxt =
'["one","two","three"]'
```

2. In the DevTools of your system browser, open the `sampleHTMLFile.html` file to view the code. Create a breakpoint in the `setup` function, after the code creates the `htmlComponent` object and adds the `"MyMATLABEvent"` listener.

3. Open the DevTools console and manually create both the JavaScript `htmlComponent` object and the event data from the MATLAB event. Use the `JSON.parse` method to convert the JSON string you just generated in MATLAB to a JavaScript object and store it in the `Data` property of the event data.

   ```
   var htmlComponent = {
       Data: '',
       addEventListener: function() {console.log("addEventListener called with: ", arguments)},
       sendEventToMATLAB: function() {console.log("sendEventToMATLAB called with: ", arguments)}
   };
   var event = {
       Data: JSON.parse('["one","two","three"]'), // JSON-formatted text from MATLAB data
   };
   ```

4. While still in the DevTools console, call the `setup` function and pass it the `htmlComponent` object you created.

   ```
   setup(htmlComponent)
   ```

5. While paused at the breakpoint, execute the `eventFromMATLABToHTML` callback function and pass it the `event` object you created to simulate an event that MATLAB sent. The **Event data** edit field updates in the HTML page.

   ```
   eventFromMATLABToHTML(event)
   ```

Simulate Sending Data from JavaScript to MATLAB

To debug how your MATLAB code reacts to component data changes from JavaScript, simulate the way that JavaScript sends component data to MATLAB.

1. In the DevTools of your system browser, open the console. Create a JavaScript object with the component data you want to simulate sending.

   ```
   var data = [[1,2],[3,4]];
   ```

2. Call the JavaScript `JSON.stringify` method to display how the component converts the data for MATLAB.

   ```
   JSON.stringify(data)
   ```

   ```
   '[[1,2],[3,4]]'
   ```

3. In the MATLAB Command Window, use the `jsondecode` function to convert this data to a MATLAB data type. The output represents the format of the data sent to MATLAB in the `Data` property of the `HTML` UI component object.

   ```
   jsondecode('[[1,2],[3,4]]')
   ```

   ```
   ans =

        1     2
        3     4
   ```

Simulate Sending Events from JavaScript to MATLAB

To debug how your MATLAB code reacts to an event from JavaScript, simulate the way that JavaScript sends event data to MATLAB.

1. In the DevTools of your system browser, open the console. Create a JavaScript object with the event data you want to simulate sending.

   ```
   var data = [[1,2],[3,4]];
   ```

2. Call the JavaScript `JSON.stringify` method to display how the component converts the data for MATLAB.

   ```
   JSON.stringify(data)
   ```

   ```
   '[[1,2],[3,4]]'
   ```

3. In the MATLAB Command Window, use the `jsondecode` function to convert this data to a MATLAB data type. The output represents the format of the data sent to MATLAB in the `HTMLEventData` property of the `HTMLEventReceivedFcn` event data.

   ```
   jsondecode('[[1,2],[3,4]]')
   ```

   ```
   ans =

        1     2
        3     4
   ```

See Also

Functions
`uihtml` | `sendEventToHTMLSource` | `jsonencode` | `jsondecode`

Properties
HTML Properties

Related Examples

- “Create HTML Content in Apps” on page 4-23
- “Display HTML Elements Styled by a Cascading Style Sheet” on page 7-15

Add Tables to App Designer Apps

To display tabular data in an App Designer app, use a table UI component. You can configure options for app users to interact with that data by sorting, selecting, or rearranging rows, columns, or cells in the app.

To add a table UI component to an App Designer app, you must work in both **Design View** and **Code View**.

Use **Design View** to:

- Create the table UI component.
- Specify row and column names.
- Specify interactivity options such as sortability and editability.

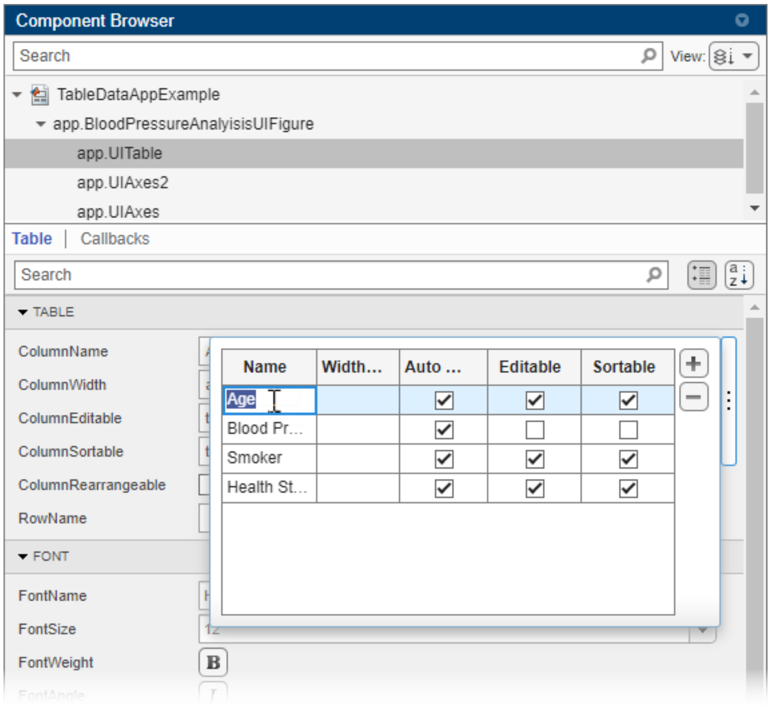
Create a `StartupFcn` callback in **Code View** to:

- Populate the table data.
- Configure the data appearance.

Create Table and Configure Table Behavior

In **Design View**, follow these steps to add a table UI component to your app:

1. Drag a **Table** component from the **Component Library** onto the app canvas.
2. Select the table UI component in the **Component Browser**.
3. To configure column information for the table, click the ⋮ button to the right of the column-related table properties. Use the editor to interactively add and rename table columns. You can also specify interactivity options for each column, such as whether the column is editable or sortable when a user interacts with the table in an app.



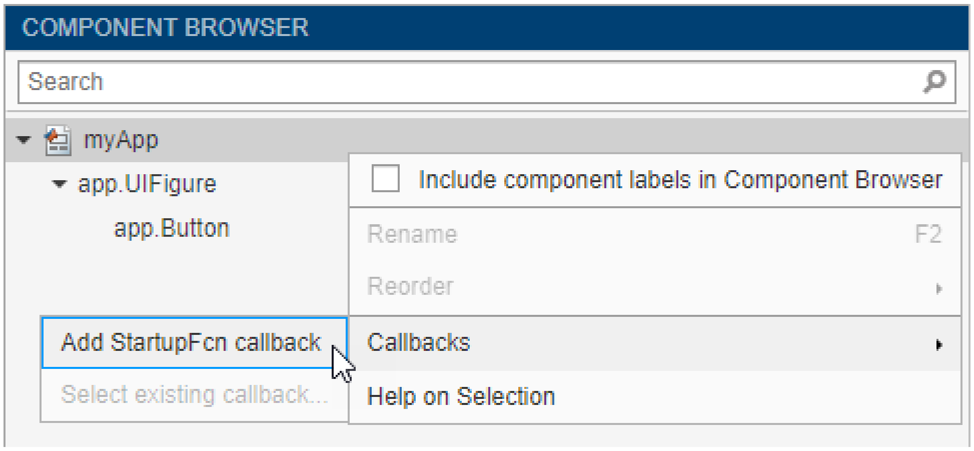
Name	Width...	Auto ...	Editable	Sortable
Age		☑	☑	☑
Blood Pr...		☑	☐	☐
Smoker		☑	☑	☑
Health St...		☑	☑	☑

4 To configure row names, use the **RowName** field in the **Component Browser**. However, the row names appear only once the table is populated with data when the app is run, and therefore do not appear in **Design View**.

Populate Table Data

In **Code View**, use these steps to populate table data in a `StartupFcn` callback. This callback is executed when a user runs the app.

1 In the **Component Browser**, right-click the app node and select **Callbacks** > **Add StartupFcn callback**. The app node has the same name as your MLAPP file.



2 In the callback function code in **Code View**, programmatically assign your table data to the table UI component using the `Data` property. For example, this code reads sample patient data and populates the table with that data.

```
function startupFcn(app)
    % Read table array from file
    t = readtable("patients.xls");
    vars = {'Age','Systolic','SelfAssessedHealthStatus','Smoker'};
    t = t(1:20,vars);

    % Add data to the table UI Component
    app.UITable.Data = t;
end
```

For more information about how table data is displayed in a table UI component, see “Display Tabular Data in Apps” on page 4-15.

3. Optionally, in the callback function code, modify the way that the table data is displayed by using `uistyle`. For example, change the background color and font color of the first column of the table by adding this code to the `StartupFcn` callback.

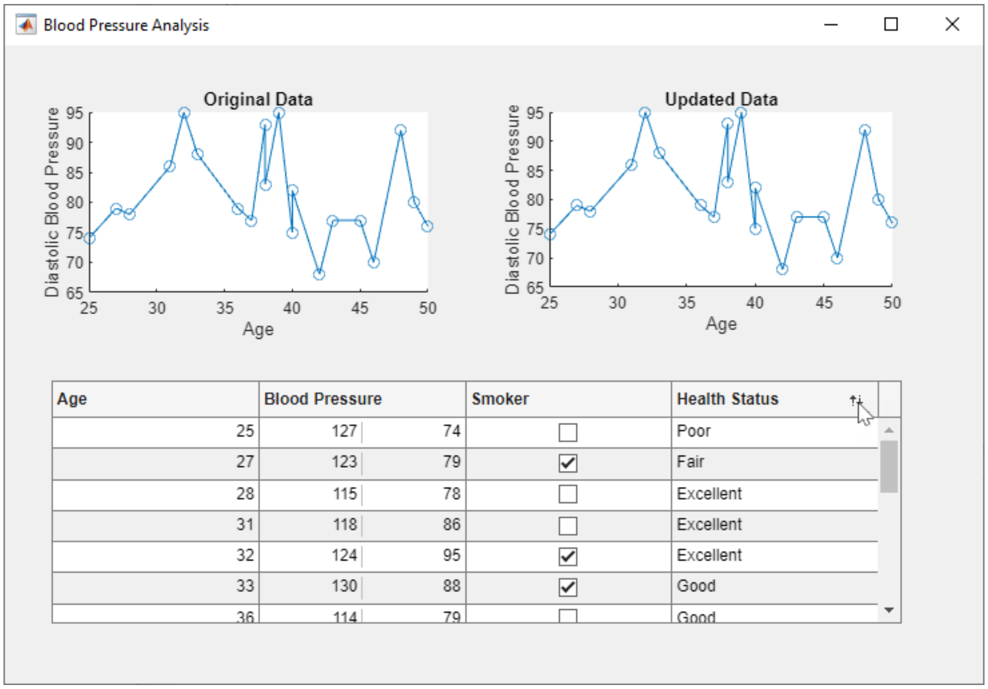
```
s = uistyle("BackgroundColor","black","FontColor","white");
addStyle(app.UITable,s,"column",1);
```

For more information, see “Style Cells in a Table UI Component” on page 15-15.

Example: App That Displays a Table

This example shows how to display a table UI component in an app that uses table data. The table contains `numeric`, `logical`, `categorical`, and multicolumn variables.

The `StartupFcn` callback loads a spreadsheet into a table array. Then a subset of the data displays and is plotted in the app. One plot displays the original table data. The other plot initially shows the same table data and then updates when the user edits a value or sorts a column in the table UI component.



Age	Blood Pressure		Smoker	Health Status
25	127	74	□	Poor
27	123	79	☑	Fair
28	115	78	□	Excellent
31	118	86	□	Excellent
32	124	95	☑	Excellent
33	130	88	☑	Good
36	114	79	□	Good

See Also

Functions

`uitable` | `uistyle` | `addStyle`

Properties

Table Properties

More About

- “Display Tabular Data in Apps” on page 4-15
- “Style Cells in a Table UI Component” on page 15-15
- “Startup Tasks and Input Arguments in App Designer” on page 6-8
- “Programmatic App That Displays a Table” on page 15-8

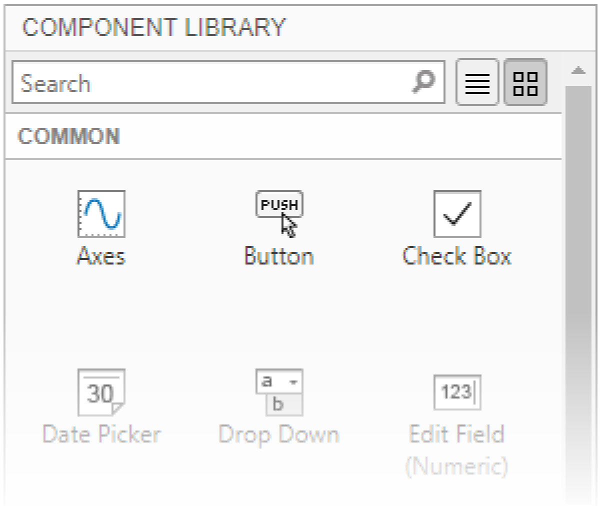
App Layout

Lay Out Apps in App Designer Design View

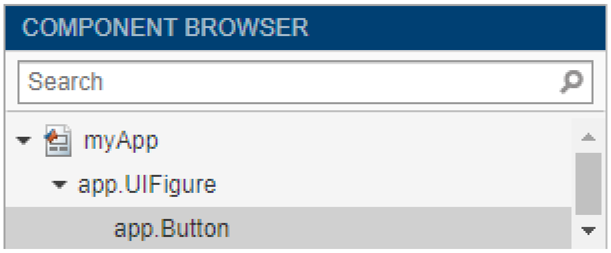
Design View in App Designer provides a rich set of layout tools for designing modern, professional-looking applications. It also provides an extensive library of UI components, so you can create various interactive features. Any changes you make in **Design View** are automatically reflected in **Code View**. Thus, you can configure many aspects of your app without writing any code.

To add a component to your app, use one of these methods:

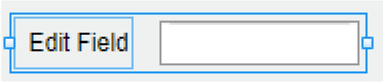
- Drag a component from the **Component Library** and drop it on the canvas.
- Click a component in the **Component Library** and then move your cursor over the canvas. The cursor changes to a crosshair. Click your mouse to add the component to the canvas in its default size, or click and drag to size the component as you add it. Some components can only be added in their default size.



The name of the component appears in the **Component Browser** after you add it to the canvas. You can select components in either the canvas or the **Component Browser**. The selection occurs in both places simultaneously.



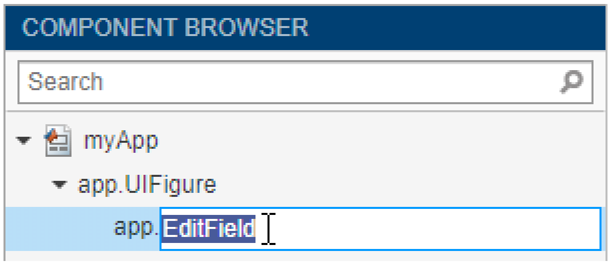
Some components, such as edit fields and sliders, are grouped with a label when you drag them onto the canvas.



These labels do not appear in the **Component Browser** by default, but you can add them to the list by right-clicking anywhere in the **Component Browser** and selecting **Include component labels in**

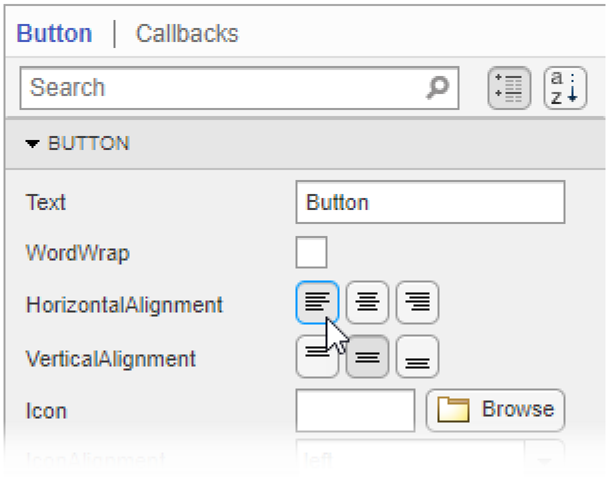
Component Browser. If you do not want the component to have a label, you can exclude it by pressing and holding the **Ctrl** key as you drag the component onto the canvas. If you want to add a label to a component without one, right-click the component and select **Add Label**.

If a component has a label, and you change the label text, the name of the component in the **Component Browser** changes to match that text. You can customize the name of the component by double-clicking it and typing a new name.



Customize Components

You can customize the appearance of a component by selecting it and then editing its properties in the component tab of the **Component Browser**. For example, from the **Button** tab you can change the alignment of the text that displays on a button.



Text Button

WordWrap ☐

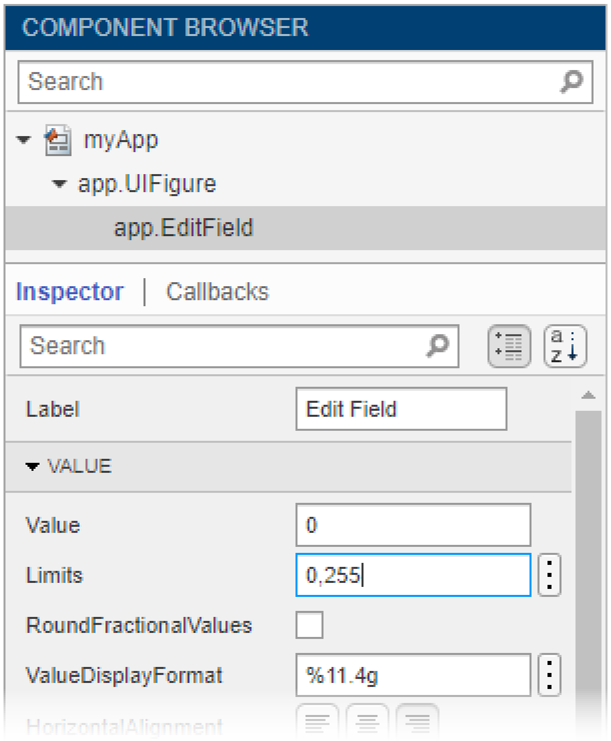
HorizontalAlignment

VerticalAlignment

Icon

IconAlignment left

Some properties control the behavior of the component. For example, you can change the range of values that a numeric edit field accepts by changing the **Limits** property.


COMPONENT BROWSER

Search

myApp

app.UIFigure

app.EditField

Inspector | Callbacks

Search

Label: Edit Field

VALUE

Value: 0

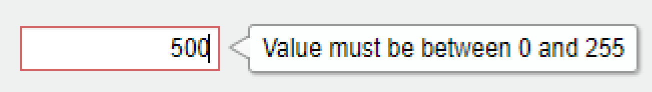
Limits: 0,255

RoundFractionalValues: □

ValueDisplayFormat: %11.4g

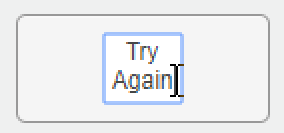
HorizontalAlignment

When the app runs, the edit field accepts values only within that range.


500

Value must be between 0 and 255

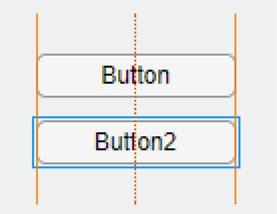
You can edit some properties directly in the canvas by double-clicking the component. For example, you can edit a button label by double-clicking it and typing the desired text. To add multiple lines of text, hold down the **Shift** key and press **Enter**.

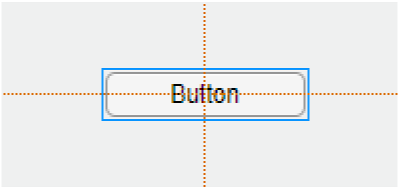

Try
Again

Align and Space Components

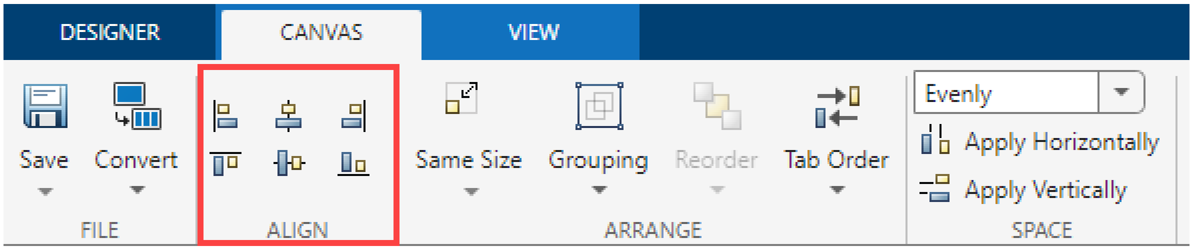
In **Design View**, you can arrange and resize components by dragging them on the canvas, or you can use the tools available in the **Canvas** tab of the toolstrip.

App Designer provides alignment hints to help you align components as you drag them in the canvas. Orange dotted lines passing through the centers of multiple components indicate that their centers are aligned. Orange solid lines at the edges indicate that the edges are aligned. Perpendicular lines indicate that a component is centered in its parent container.

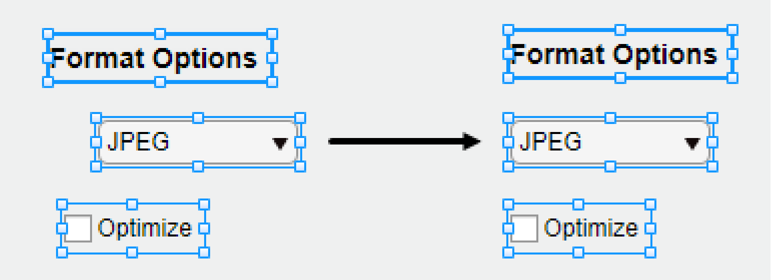




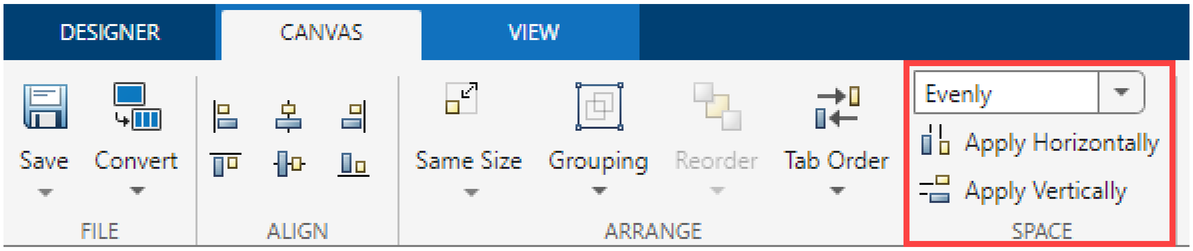
As an alternative to dragging components on the canvas, you can align components using the tools in the **Align** section of the toolstrip.



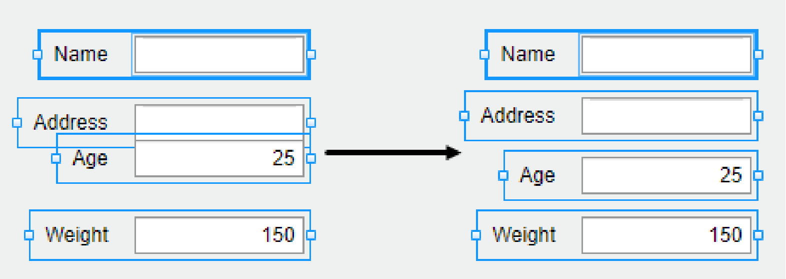
When you use an alignment tool, the selected components align to an anchor component. The anchor component is the last component selected, and it has a thicker selection border than the other components. To select a different anchor, hold down the **Ctrl** or **Shift** key and click the desired component twice (once to deselect the component, and a second time to select it again). For example, in the following image, the **Format Options** label is the anchor. Clicking the **Align left** button aligns the left edges of the drop-down and check box to the left edge of the label.



You can control the spacing among neighboring components using the tools in the **Space** section of the toolstrip. Select a group of three or more components, and then select an option from the drop-down list in the **Space** section of the toolstrip. The **Evenly** option distributes the space evenly within the space occupied by the components. The **20** option spaces the components 20 pixels apart. If you want to customize the number of pixels between the components, type a number into the drop-down list.



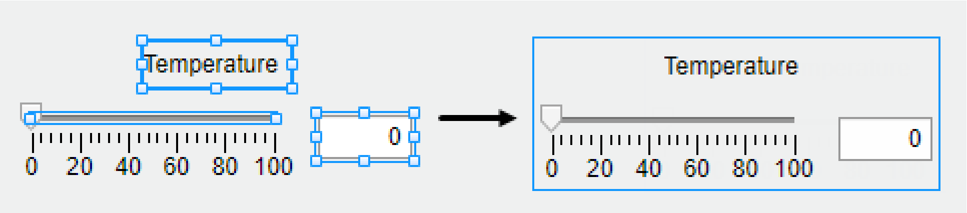
Next, click **Apply Horizontally** or **Apply Vertically** . For example, select **Evenly** and then click **Apply Vertically** to distribute the space among a vertical stack of components.



Group Components

You can group two or more components together to modify them as a single unit. For example, you can group a set of components after finalizing their relative positions, so you can then move them without changing that relationship.

To group a set of components, select them in the canvas, and then select **Grouping** > **Group** in the **Arrange** section of the toolstrip.



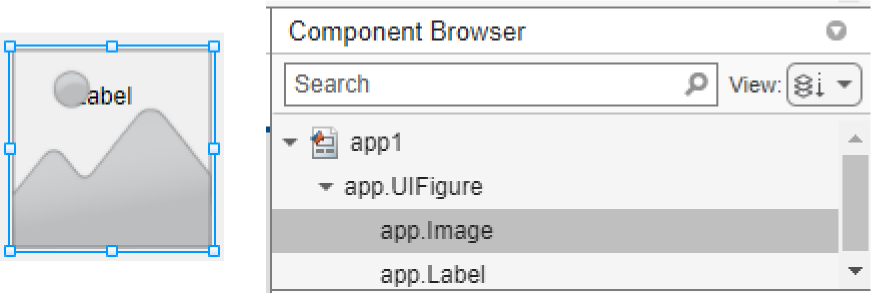
The **Grouping** tool also provides functionality for these common tasks:

- Ungroup all components in a group — Select the group. Then select **Grouping** > **Ungroup**.
- Add a component to a group — Select the component and the group. Then select **Grouping** > **Add to Group**.
- Remove a component from a group — Select the component. Then select **Grouping** > **Remove from Group**.

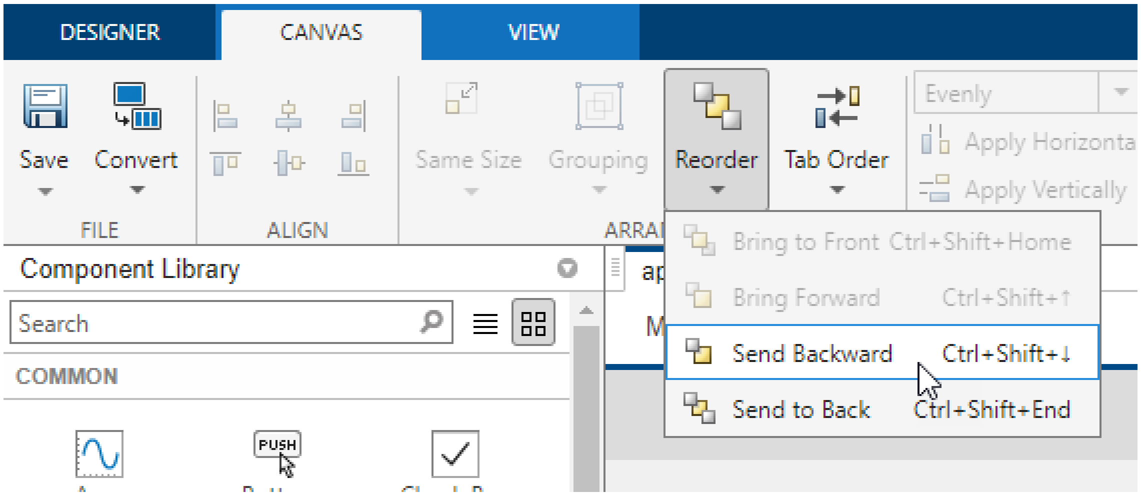
Reorder Components

You can change the order in which components stack on top of each other by using the **Reorder** tool in **Design View**.

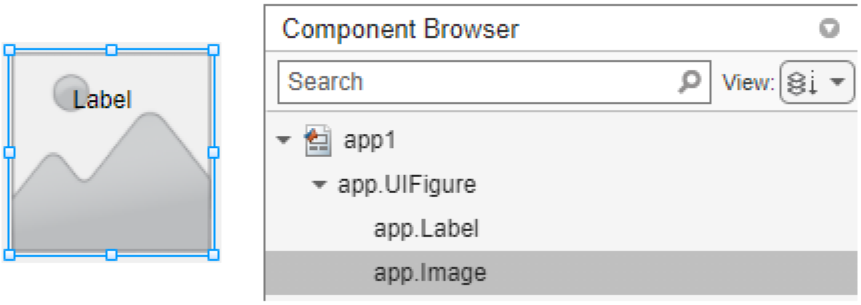
For example, create a label and then create an image. By default, the image appears on top of the label. The default view of the **Component Browser** shows the components based on their stacking order, with the image first since it is on top and the label second.



To reorder the components so that the label is on top of the image, select the image on the canvas, and then select **Reorder** in the toolstrip. You can also right-click the image and select the **Reorder** tool. Send the image backward by choosing **Send Backward**.

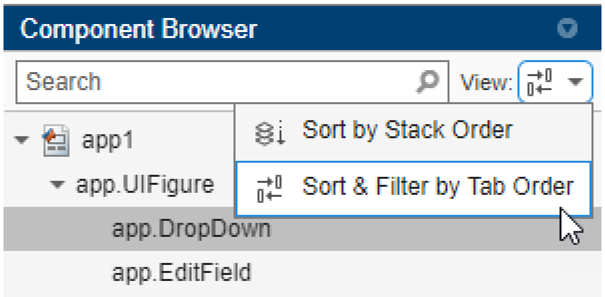


The image now is behind the label. When you reorder components, the order of the components inside the **Component Browser** also changes.

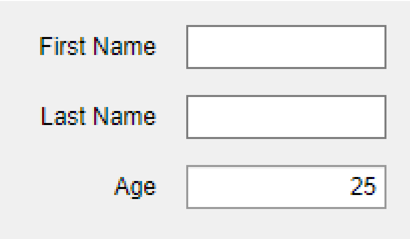


Modify Tab Focus Order of Components

When users run your app, they can use the **Tab** key to navigate between app components. To view the order in which the components come into focus when a user presses **Tab**, expand the **View** drop-down list in the **Component Browser** and select **Sort & Filter by Tab Order**. The **Component Browser** lists only the components in the app that can have focus, in the order of focus. You can change the tab order of the components by clicking and dragging the component names in the **Component Browser**.



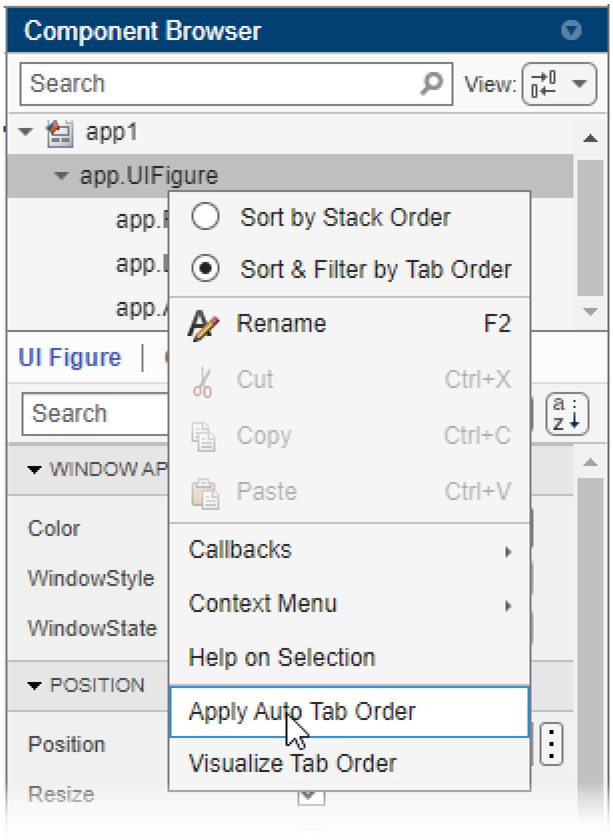
Alternatively, App Designer can automatically apply a left-to-right and then top-to-bottom tab focus order for components. Right-click the name of the container in the **Component Browser** and select **Apply Auto Tab Order**. For example, in an app with stacked edit fields for app users to enter their first name, last name, and age, right-click the `app.UIFigure` node in the **Component Browser** and apply automatic tab ordering. When users the app, they can use the **Tab** key to navigate between the edit fields and enter a first name, then a last name, and finally an age.



First Name

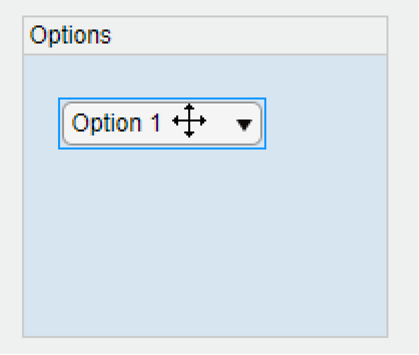
Last Name

Age 25

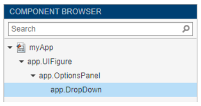


Arrange Components in Containers

When you drag a component into a container such as a panel, the container turns blue to indicate that the component is a child of the container. This process of placing components into containers is called parenting.



The **Component Browser** shows the parent–child relationship by indenting the name of the child component under the parent container.

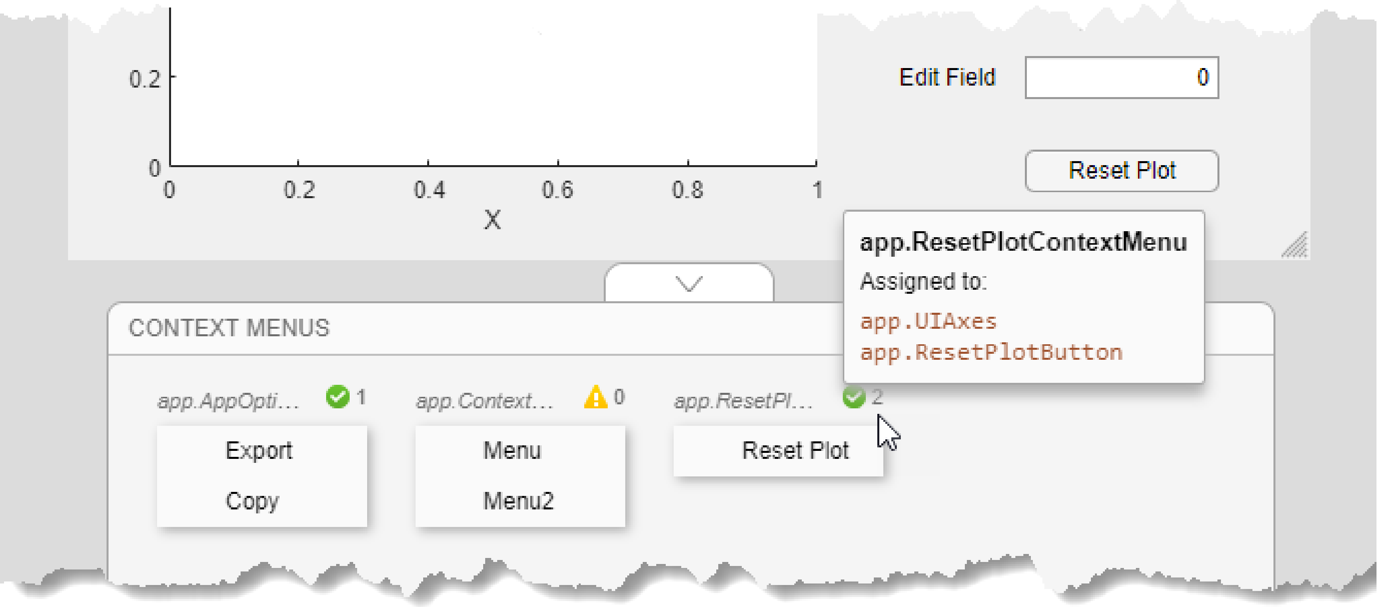


Create and Edit Context Menus in App Designer

There are several ways to create context menus in App Designer. Since context menus are visible only when you right-click a component in the running app, they do not appear in the figure when you are in **Design View**. This makes the workflow for editing context menus slightly different than for other components. These sections describe the ways to create and edit context menus.

Create Context Menus

To create a context menu, drag it from the **Component Library** onto the UI figure or another component. This assigns the context menu to the `ContextMenu` property of that component. When you create a context menu it appears in an area on the canvas below the figure. This **Context Menus** area gives you a preview of each context menu you created and indicates how many components each one is assigned to. For example, this is how one set of context menus might appear on the canvas:



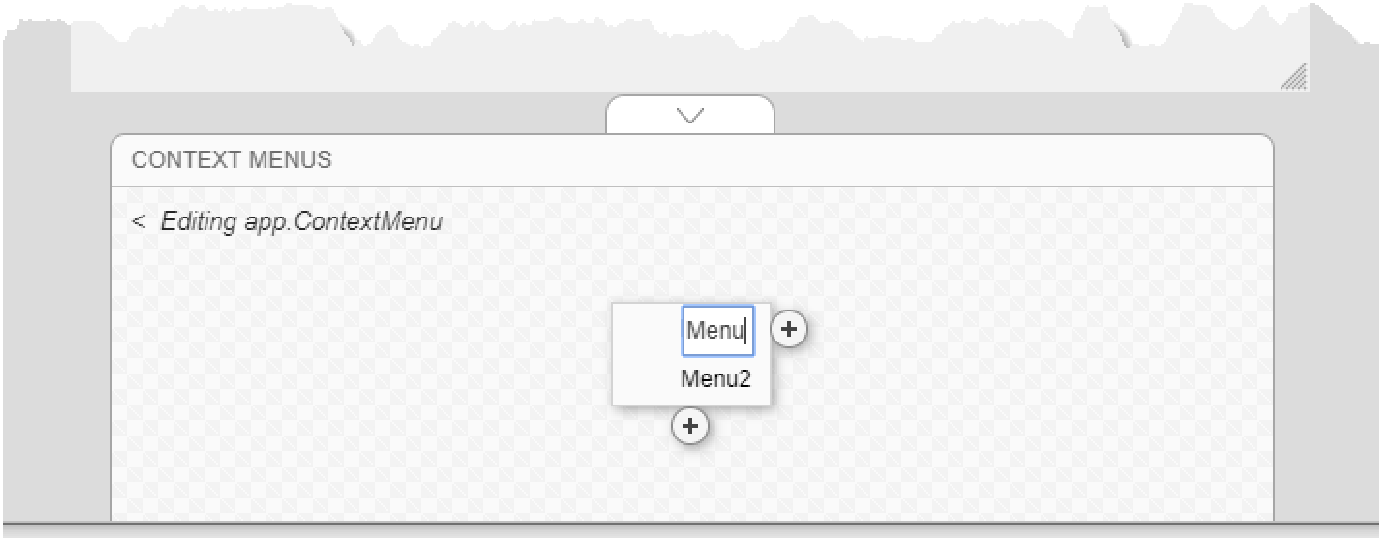
If you want to create a context menu without assigning it to a component, drag it to the **Context Menus** area instead.

Alternatively, create and assign a context menu to a specific component by right-clicking on that component and selecting **Context Menu > Add New Context Menu**.

All context menus are created as children of the UI figure and are added to the **Component Browser**, even if they are not assigned to a component.

Edit Context Menus

Edit a context menu by double-clicking it in the **Context Menus** area or by right-clicking it and selecting the edit option for the name of your menu. This brings the context menu into the **Context Menus** editing area where you can edit and add menu items and submenus.



When you are finished editing, click the back arrow (**<**) to exit the edit area.

Change Context Menu Assignments

To disassociate a context menu from a component, right-click on the component and select **Context Menu** > **Unassign Context Menu**.

To replace the context menu that is assigned to a component with another one, you can drag the context menu onto the component, or you can right-click on the component, click **Context Menu** > **Replace With**, and select one of the other context menus you have created. If you only created one context menu, then the **Replace With** option does not appear.

Alternatively, select a component in the **Component Browser** and select **Interactivity** from the component tab. Then, expand the **ContextMenu** drop-down list and select a different context menu to assign to the component.

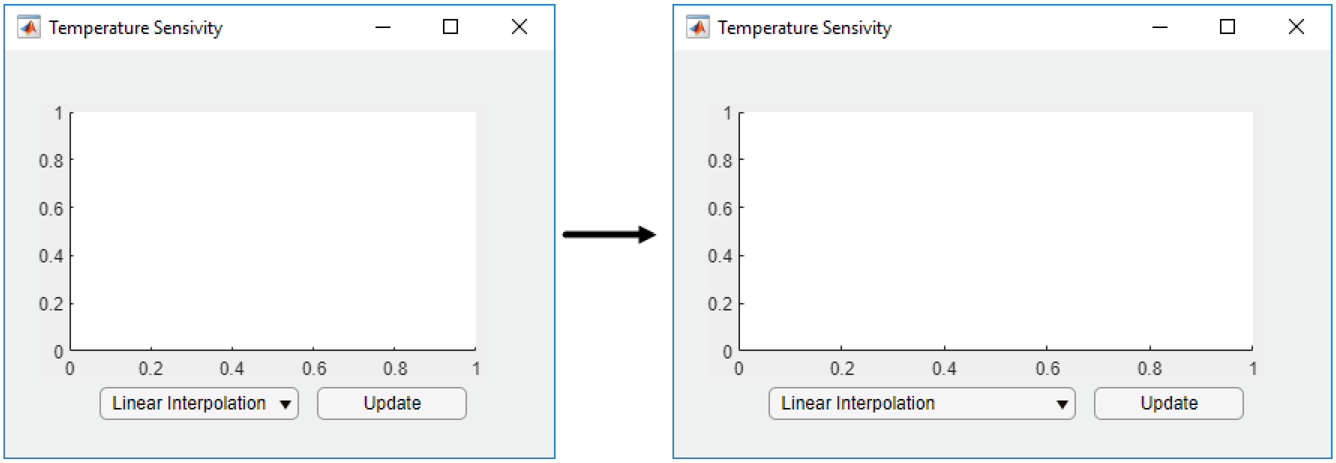
See Also

More About

- "App Building Components" on page 4-2
- "App Designer Keyboard Shortcuts" on page 9-2
- "Manage Resizable Apps in App Designer" on page 5-12

Manage Resizable Apps in App Designer

Apps you create in App Designer are resizable by default. The components reposition and resize automatically as the user changes the size of the window at run-time. The `AutoResizeChildren` property controls this automatic resize behavior. By default, App Designer enables this property for the UI figure and all its child containers such as panels and tabs. To set the `AutoResizeChildren` property of a child container to a different value, set the value for the child container after setting the value for the parent.



When the `AutoResizeChildren` property is enabled for a container, MATLAB manages the size and position of only the immediate children in the container. Components in nested containers are managed by the `AutoResizeChildren` property of their immediate parent. To ensure that the alignment of components relative to one another (like a grouping of buttons) is preserved when your app is resized, parent the grouping of components to a panel, instead of directly to the figure.

Resizing Graphics Objects with Normalized Position Units

When graphics objects, like axes or charts, use normalized position units and are the child of a resizable container, certain properties of the graphics object are affected after the parent container is resized. For example, if axes or charts use a value of `'normalized'` for the `Units` property and are parented to a container with the `AutoResizeChildren` property set to `'on'`, then:

- The value of the `OuterPosition` property for the axes or chart changes when the app is resized.
- The axes or chart does not shrink smaller than a minimum size when the app is resized.

If you want to avoid either of these behaviors, set the `AutoResizeChildren` property of the container to `'off'`.

Alternatives to Default Auto-Resize Behaviors

If you want more flexibility over how your app automatically resizes, use grid layout managers or the auto-reflow options in App Designer instead of the `AutoResizeChildren` property. For more information about these options, see:

- “Use Grid Layout Managers in App Designer” on page 5-14

- “Apps with Auto-Reflow” on page 5-18

If the resize behaviors supported by `AutoResizeChildren`, grid layout managers, or auto-reflow options are not the behaviors you want, then you can create custom resize behaviors by writing a `SizeChangedFcn` callback function for the container. For more information, see “Manage App Resize Behavior Programmatically” on page 10-10.

See Also

UI Figure

More About

- “Lay Out Apps in App Designer Design View” on page 5-2
- “Callbacks in App Designer” on page 6-16

Use Grid Layout Managers in App Designer

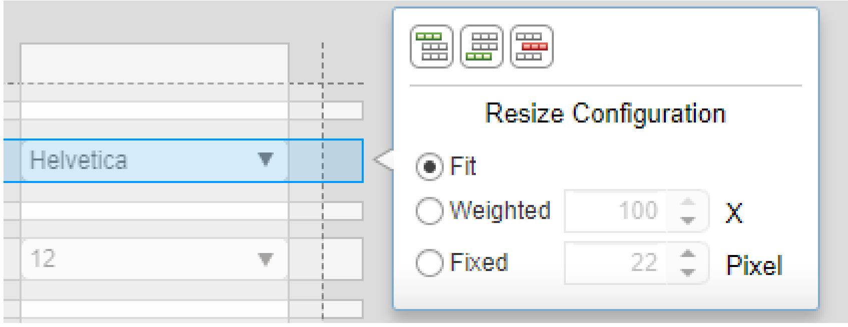
Grid layout managers provide a way to lay out your app without having to set pixel positions of UI components in `Position` vectors. For resizable apps, grid layout managers provide more flexibility than the automatic resize behavior in App Designer. They are also easier to configure than it is to code `SizeChangedFcn` callback functions.

Add and Configure Grid Layout Manager

In App Designer, you can add a grid layout manager to a blank app or to empty container components within the figure.

To use a grid layout manager, drag a grid layout from the **Component Library** onto the canvas. Alternatively, you can right-click the figure or container and select **Apply Grid Layout** from the context menu. A grid layout manager spans the entire app window or container that you place it in. It is invisible unless you are actively configuring it on the App Designer canvas.

To configure the grid layout manager, in **Design View**, bring the grid layout into focus by clicking in the area where you added it. Then, select the button from the upper-left corner of the grid layout manager, or right-click the grid layout and select **Configure Grid Layout**. Then, select a row or column and from the **Resize Configuration** menu, specify **Fit**, **Weighted**, or **Fixed**. For more information about these options, see GridLayout Properties. You can also add or remove rows and columns.



Resize Configuration

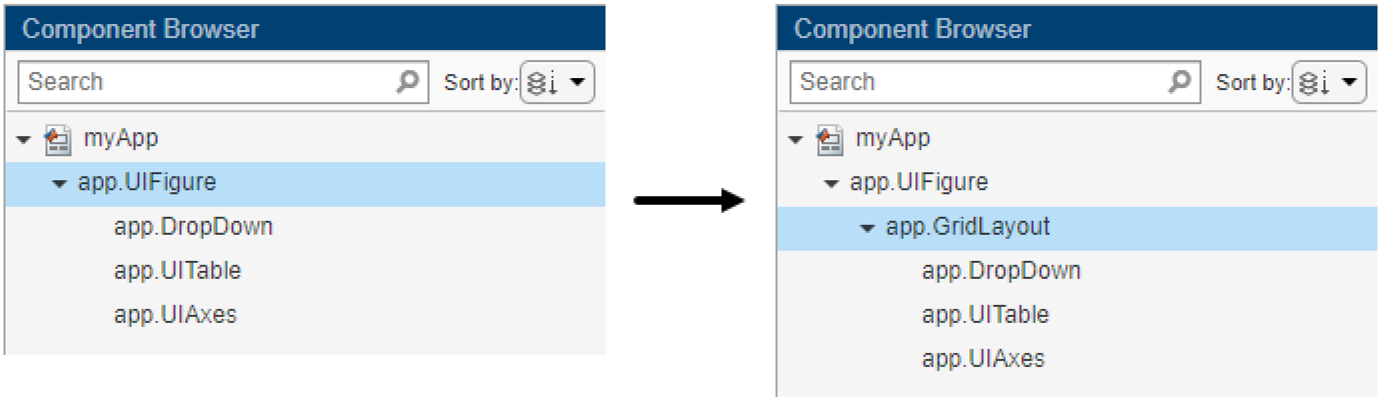
● Fit

○ Weighted 100 X

○ Fixed 22 Pixel

Convert Components from Pixel-Based Positions to Grid Layout Manager

You can convert the components within a UI figure or container from pixel-based positioning to a grid layout manager. When you apply a grid layout manager to a UI figure or container that has components in it, the components get added to the grid layout manager and their `Position` vectors get replaced by `Layout.Row` and `Layout.Column` values that specify their location in the grid. The component hierarchy also updates in the **Component Browser**.



Grid layout managers support different properties than other container components. In some cases, you might need to update your callback code if it sets these types of properties, or if it sets component properties that are not available when they are managed by the grid layout. If your callbacks or other behaviors do not work as expected, then look for code patterns like the ones lists in this table.

Symptom or Warning	Explanation	Suggested Action
`Warning: Unable to set 'Position', 'InnerPosition', or 'OuterPosition' for components in 'GridLayout'.`	You cannot set the `Position` property on components in a grid layout manager.	Specify a grid location for the component by setting the `Layout` property with appropriate `Row` and `Column` values.

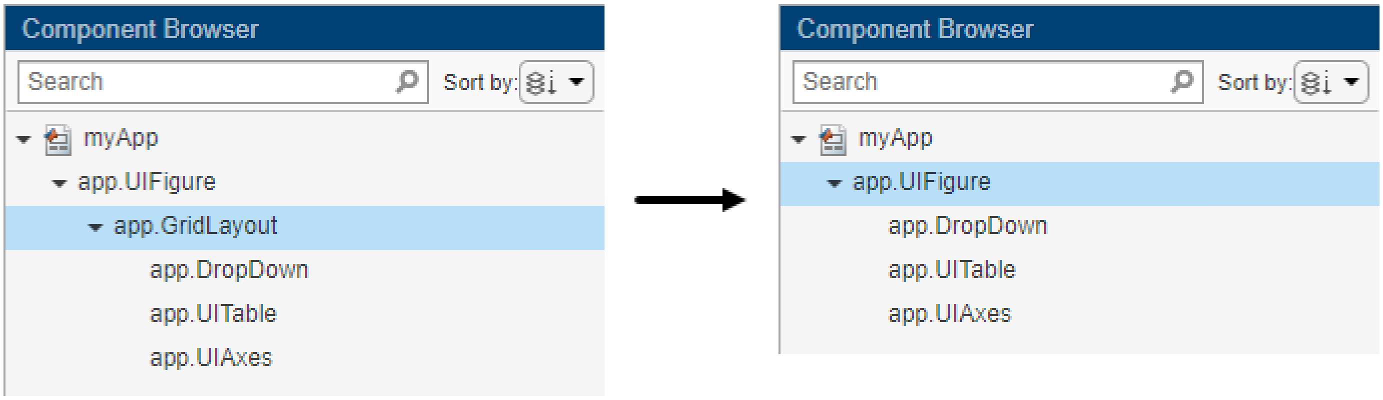
`Error using matlab.ui.container.GridLayout/set`
`There is no FontSize property on the GridLayout class.`
	Properties you set on other container components might not be supported on the grid layout manager.	Update your code so that it sets properties on the intended container.

A context menu assigned to a container does not open in the running app.	When you add a grid layout manager to a container, it spans the entire container. This means that click events happen on the grid, instead of the container.	Reassign the context menu to the grid layout.

Convert Components from Grid Layout Manager to Pixel-Based Positions

Starting in R2022a, you can also remove a grid layout manager from your app and convert the components in the grid to use pixel-based positioning. To remove a grid layout manager from a container, right-click the container in the canvas and select **Remove Grid Layout**.

When you remove a grid layout manager that has components in it from a UI figure or container, the grid layout manager is deleted and the components get added to the container that originally contained the grid layout manager. The `Layout.Row` and `Layout.Column` values that specified the component locations in the grid get replaced by `Position` vectors. The component hierarchy also updates in the **Component Browser**.

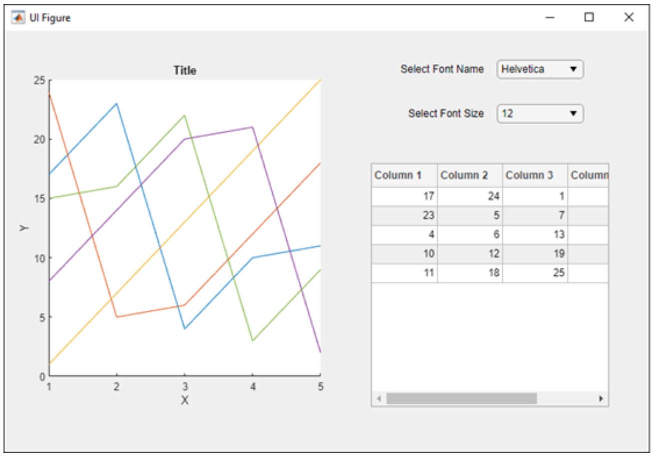


In some cases, you might need to update your callback code if it sets properties of the removed grid layout manager.

Example: Convert Components to Use Grid Layout Manager Instead of Pixel-Based Positions

This app shows how to apply a grid layout manager to the figure of an app that already has components in it. It also shows how to configure the grid layout manager so that the rows and columns automatically adjust to accommodate changes in size of text-based components.

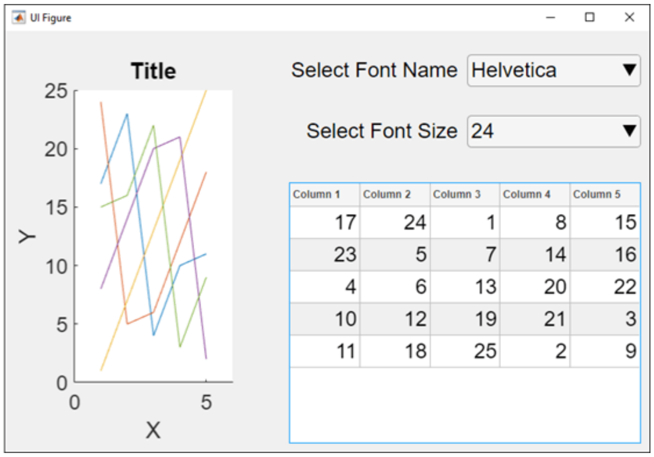
1. Open the app in App Designer. In **Design View**, drag a grid layout manager into the figure.
2. Right-click the grid layout manager that you just added to the figure and select **Configure Grid Layout** from the context menu.
3. One-by-one, select the rows and columns of the grid that contain the drop-down menus and the table and change their resize configurations to **Fit**. When you are finished, verify that in the **Inspector** tab of the **Component Browser**, the **ColumnWidth** values are `12.64x,1.89x,fit,fit,fit,fit` and the **RowHeight** values are `1x,fit,1.93x,fit,3.07x,fit`.
4. Switch to **Code View**. Update each of the `DropDownValueChanged` callbacks so that the `allchild` functions set the font name and font size on components in `app.GridLayout`, instead of in `app.UIFigure`.
5. Now run the app to see how the grid adjusts to accommodate the components as their sizes change.



Select Font Name Helvetica

Select Font Size 12

Column 1	Column 2	Column 3	Column
17	24	1	
23	5	7	
4	6	13	
10	12	19	
11	18	25	



Select Font Name Helvetica

Select Font Size 24

Column 1	Column 2	Column 3	Column 4	Column 5
17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

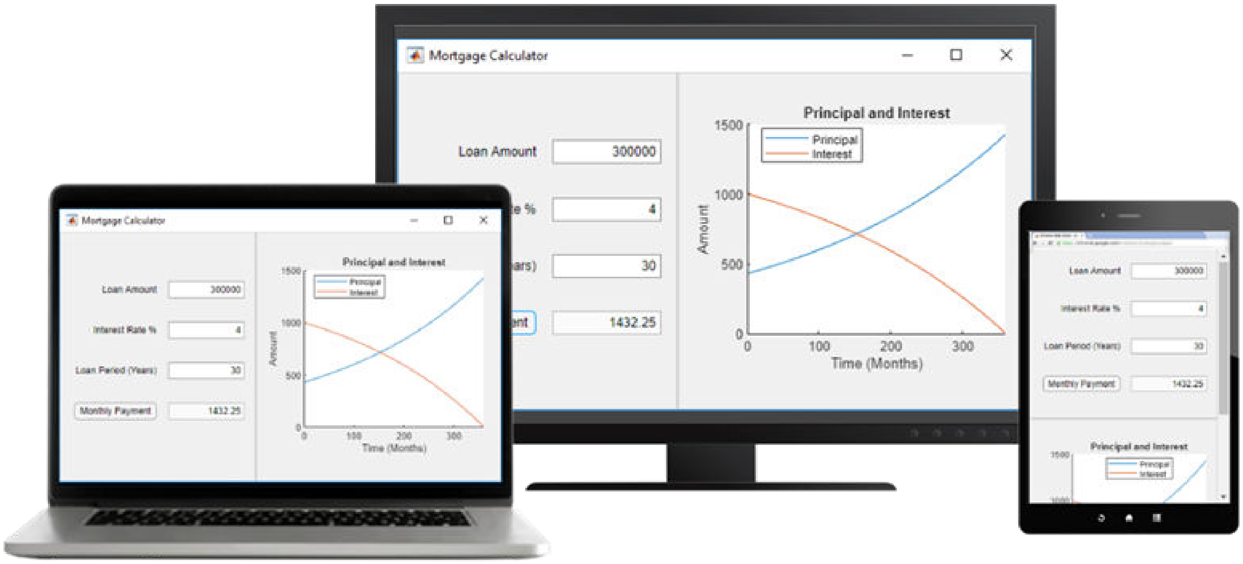
See Also

Functions
`uigridlayout`

Properties
GridLayout Properties

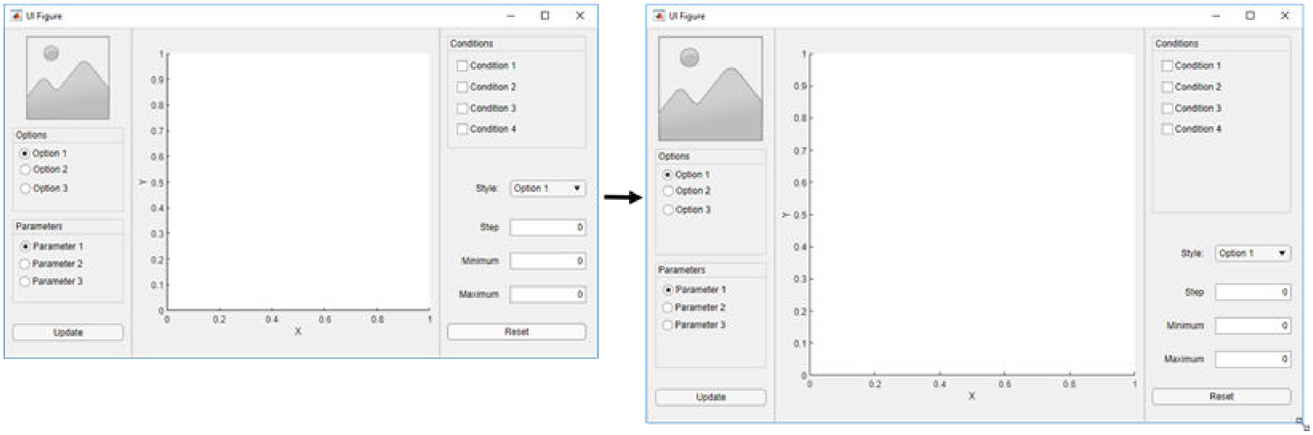
Apps with Auto-Reflow

Apps with auto-reflow are preconfigured app types that optimize the viewing experience by automatically adjusting the size, location, and visibility of the app content in response to screen size, orientation, and platform. Use apps with auto-reflow if you expect to run or share your apps across multiple environments or desktop resolutions.

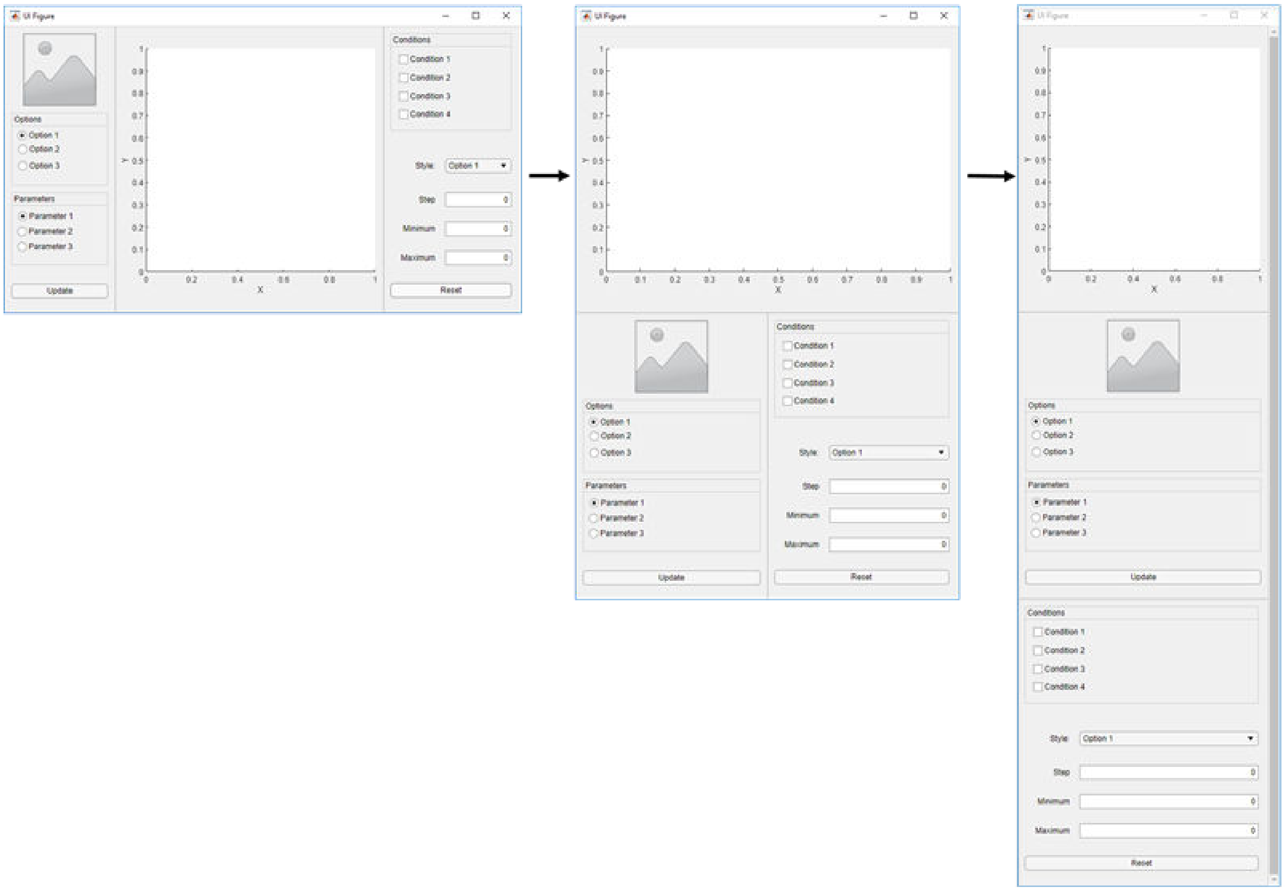


What Is Auto-Reflow?

Apps with auto-reflow extend the existing auto-resize behaviors that are on by default in all App Designer apps. These apps detect and adapt to the available screen size when they are first displayed. Both 2- and 3-panel apps have a large flexible-size panel, intended for visualizations like plots. As the app changes size, the large panel grows or shrinks, depending on the space available.



When an app is resized beyond a certain predefined threshold, the panels in the app reflow and reorder to make the best use of the space. As panels reorder themselves, they and the components in them dynamically adjust in size while extra space between components (white space) is also reduced.



When an app becomes very small, auto-resize stops eliminating whitespace and resizing components. This can put some components outside the visible part of the window. To access these components, set the `Scrollable` property of the panels to `'on'`. This enables scroll bars to appear when necessary.

Create New App with Auto-Reflow

The App Designer Start Page includes options to create new 2-panel and 3-panel apps with auto-resize and auto-reflow, and canvas interactions to guide app building. No additional code is needed to achieve the reflowing and resizing behavior.

Convert Existing App to Use Auto-Reflow

You can also convert an existing app into an app with auto-reflow by expanding the **Convert** drop-down list from the **File** section of the **Canvas** tab and selecting `2-Panel App with Auto-Reflow` or `3-Panel App with Auto-Reflow`.

When you convert an existing app to an app with auto-reflow, App Designer:

- Creates a duplicate of your app with `_converted` appended to the file name. Your original app file is not changed.

- Automatically adds preconfigured panels and a grid layout to your app to provide the automatic reflow and resize behaviors.
- Creates a `SizeChangedFcn` callback function in order to control the layout of the app as the figure is resized.

In some cases, after App Designer has converted your app, you may need to update your callback code or the position of some components. This table describes some examples of adjustments you that you may need to make.

Symptom	Explanation	Suggested Action
Components overlap	App Designer tries to maintain the relative positions of your components, but you may need to make some minor adjustments.	Adjust the position of components as needed.
Callback code does not behave as expected	When the preconfigured panels are added to your app the hierarchy of the components in your app changes. If your callbacks reference components based on their parent, they may need to be updated.	Update the parent of the components in your callbacks.
Existing `SizeChangedFcn` callback on the UI figure does not behave as expected	Apps with auto-reflow generate their own `SizeChangedFcn` callback for the figure. If your app already had a `SizeChangedFcn` callback for the figure, App Designer disconnects it from the figure, but it does not remove the code.	After your app has been converted, modify or remove the `SizeChangedFcn` callback that was disconnected from the figure. You can assign it to another container component, or remove it if it is no longer needed.

Remove Auto-Reflow Behavior

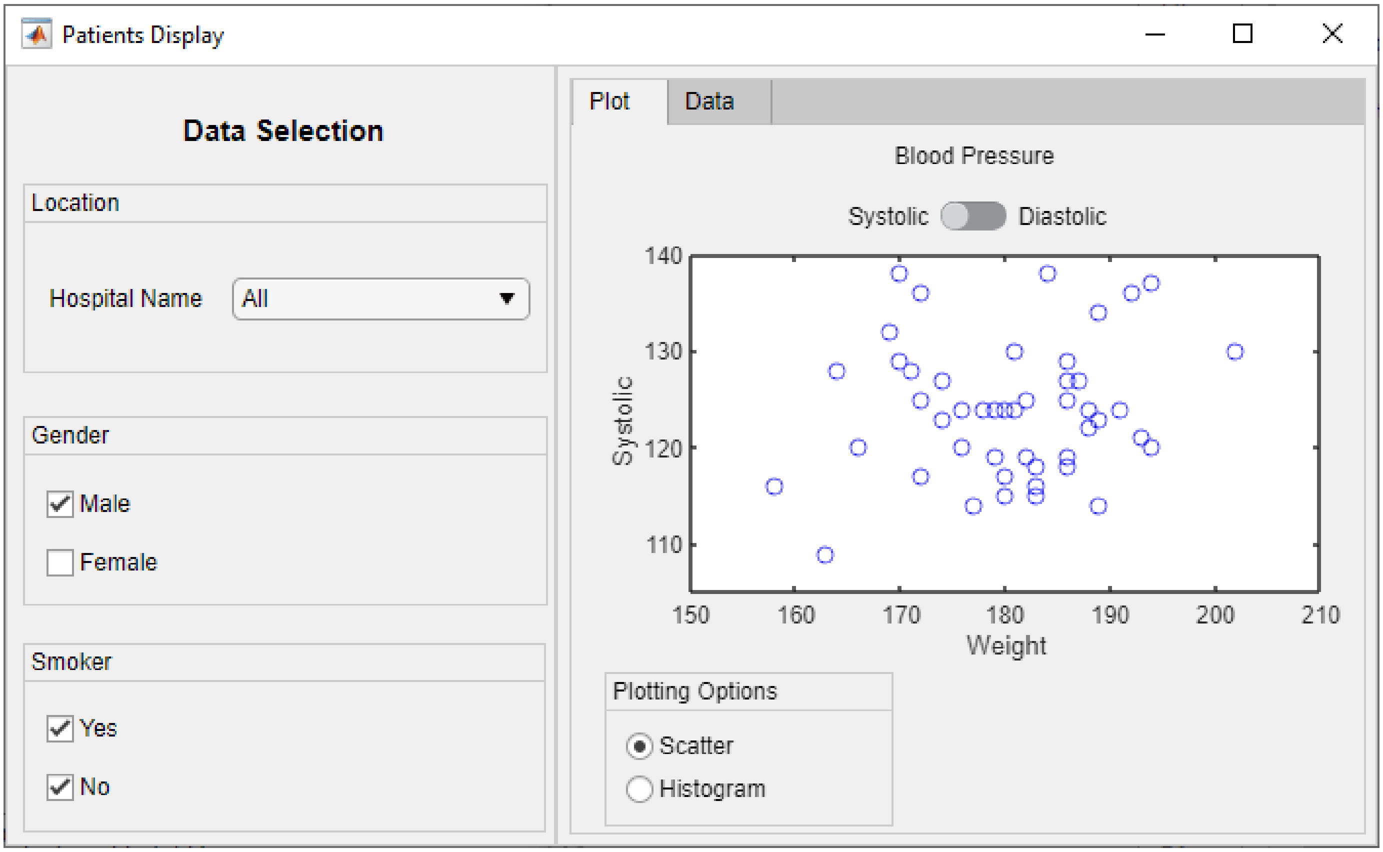
You can remove auto-reflow behavior from an existing app by expanding the **Convert** drop-down list from the **File** section of the **Canvas** tab and selecting `App without Auto-Reflow`.

When you convert an app with auto-reflow to an app without auto-reflow, App Designer:

- Creates a duplicate of your app with `_converted` appended to the file name. Your original app file is not changed.
- Removes the preconfigured grid layout from the app with auto-reflow.
- Removes the `SizeChangedFcn` callback function that is used to control the layout of the app with auto-reflow.

Example: App with Auto-Reflow

This app has components within panels that have auto-reflow behavior. Controls for data selection are parented to the left panel and data visualizations are parented to two tabs in the right panel. Run the app and change the size of the app window. The app content resizes and reflows based on the app window size.



Patients Display

Data Selection

Location

Hospital Name All ▼

Gender

☑ Male

□ Female

Smoker

☑ Yes

☑ No

Plotting Options

● Scatter

○ Histogram

See Also

App Designer

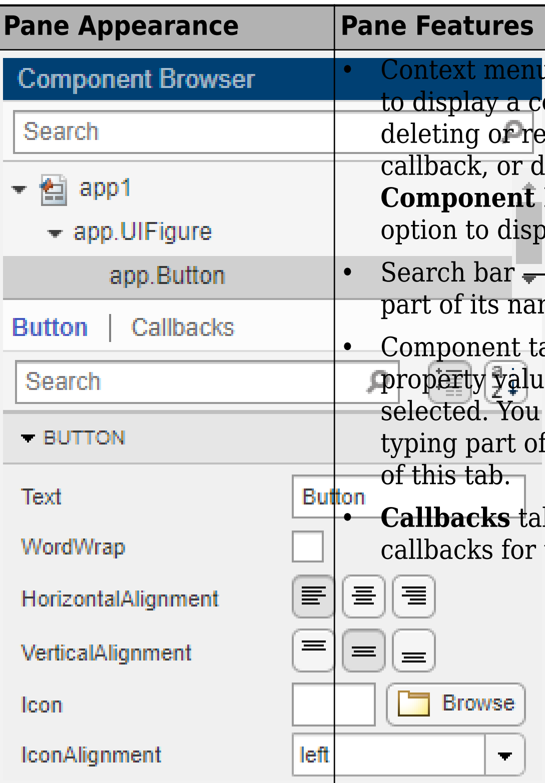
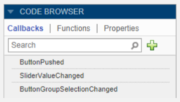
App Programming

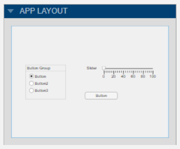
Manage Code in App Designer Code View

Code View provides most of the same programming features that the MATLAB Editor provides. It also provides a rich set of features that help you to navigate your code and avoid many tedious tasks. For example, you can search for a callback by typing part of its name in a search bar. Clicking a search result scrolls the editor to the definition of that callback. And if you change the name of a callback, App Designer automatically updates all references to it in your code.

Manage Components, Functions, and Properties

Code View has three panes to help you manage different aspects of your code. This table describes each of them.

Pane Name	Pane Appearance	Pane Features
Component Browser	 Component Browser Search app1 app.UIFigure app.Button Button | Callbacks Search BUTTON Text: Button WordWrap HorizontalAlignment VerticalAlignment Icon Browse IconAlignment: left	• Context menu — Right-click a component in the list to display a context menu that has options for deleting or renaming the component, adding a callback, or displaying help. Select the **Include Component Labels in Component Browser** option to display grouped component labels. • Search bar — Quickly locate a component by typing part of its name in the search bar. • Component tab — Use this tab to view or change property values for the component that is currently selected. You can also search for a property by typing part of the name in the search bar at the top of this tab. • **Callbacks** tab — Use this tab to manage the callbacks for the component that is selected.
Code Browser	 CODE BROWSER Callbacks | Functions | Properties Search ButtonPushed SliderValueChanged ButtonGroupSelectionChanged	• **Callbacks**, **Functions**, and **Properties** tabs — Use these tabs to add, delete, or rename any of the callbacks, helper functions, or custom properties in your app. Clicking an item in the **Callbacks** or **Functions** tab scrolls the editor to the corresponding section in your code. Rearrange the order of callbacks by selecting the callback you want to move and then, drag and drop the callback into its new position in the list. This also repositions the callback in the editor. • Search bar — Quickly locate a callback, helper function, or property by typing part of its name in the search bar.

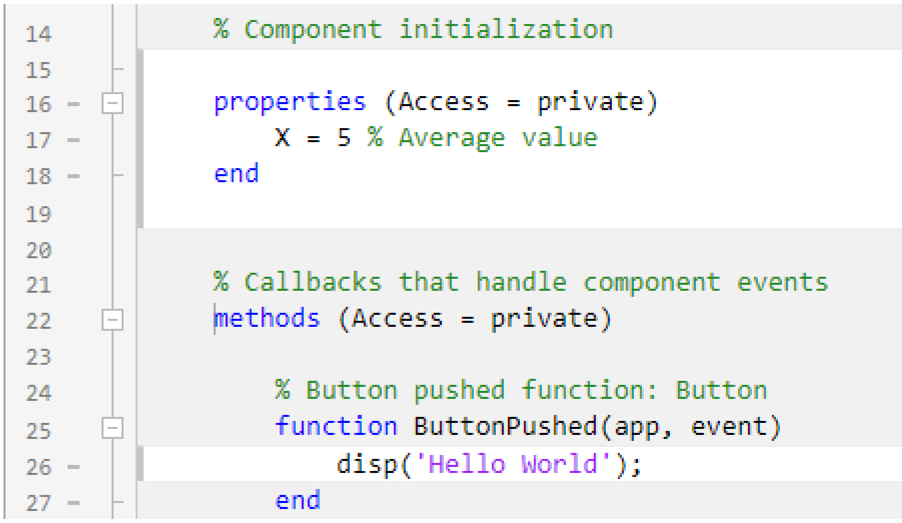
Pane Name	Pane Appearance	Pane Features
App Layout		• App thumbnail — Use the thumbnail image to locate components in large, complex apps that have many components. Selecting a component in the thumbnail selects the component in the **Component Browser**.

Identify Editable Sections of Code

In the **Code View** editor, some sections of code are editable and some are not. Uneditable sections are generated and managed by App Designer, whereas editable sections correspond to:

- The body of functions you define (e.g., callbacks and helper functions)
- Custom property definitions

In the default color scheme, uneditable sections of code are gray and editable sections of code are white.



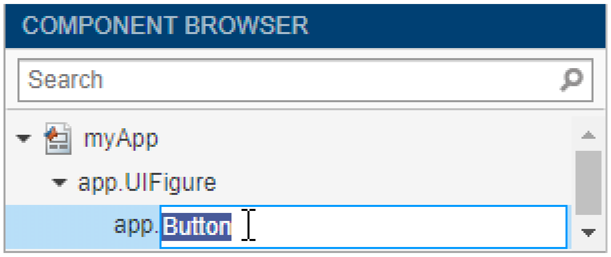
```
    % Component initialization

    properties (Access = private)
        X = 5 % Average value
    end


    % Callbacks that handle component events
    methods (Access = private)

        % Button pushed function: Button
        function ButtonPushed(app, event)
            disp('Hello World');
        end
```

Program Your App

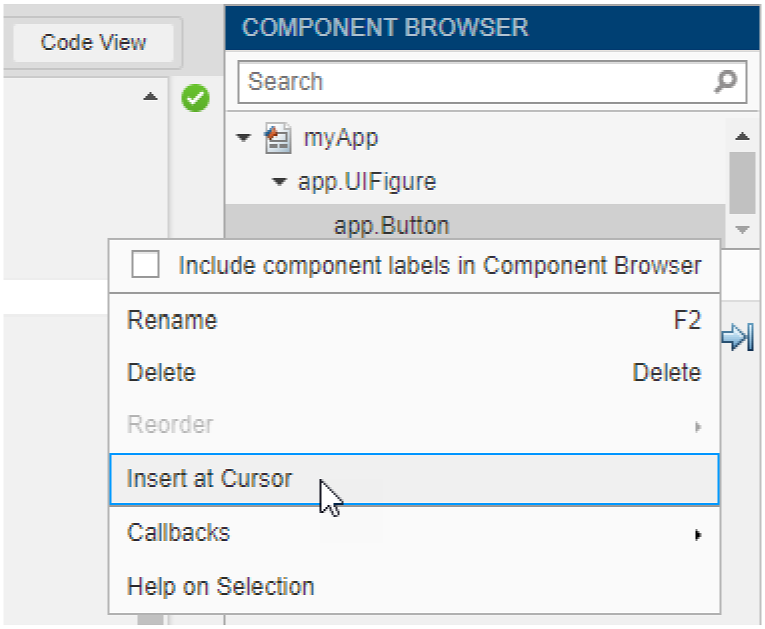
App Designer defines your app as a MATLAB class. You do not need to understand classes or object-oriented programming to create an app because App Designer manages those aspects of the code. However, programming in App Designer requires a different workflow than working strictly with functions. You can review a summary of this workflow at any time by clicking the **Show Tips** button in the **Resources** tab of the toolstrip.

Manage UI Components

When you add a UI component to your app, App Designer assigns a default name to the component. Use that name (including the app prefix) to refer to the component in your code. You can change the name of a component by double-clicking the name in the **Component Browser** and typing a new name. App Designer automatically updates all references to that component when you change its name.



To use the name of a component in your code, you can save some time by copying the name from the **Component Browser**. Place your cursor in an editable area of the code where you want to add the component name. Then, from the **Component Browser**, right-click the component name and select **Insert at Cursor**. Alternatively, you can drag the component name from the list into your code.



To delete a component, select its name in the **Component Browser** and press the **Delete** key.

Manage Callbacks

To make a component respond to user interactions, add a callback. Right-click the component in the **Component Browser** and select **Callbacks > Add (callback property) callback**.

If you delete a component from your app, App Designer deletes the associated callback only if the callback has not been edited and is not shared with other components.

To delete a callback manually, select the callback name in the **Callbacks** tab of the **Code Browser** and press the **Delete** key.

For more information about callbacks, see “Callbacks in App Designer” on page 6-16.

Share Data Within Your App

To store data, and share it among different callbacks, create a property. For example, if want your app to read a data file and allow different callbacks in your app to access that data, store the data in a property when you load the file.

To create a property, expand the **Property** drop-down in the **Editor** tab, and select **Private Property** or **Public Property**. App Designer creates a template property definition and places your cursor next to that definition. Change the name of the property as desired.

```
properties (Access = public)
        X % Average cost
end
```

To reference the property in your code, use the syntax `app.`*`PropertyName`*. For example, `app.X` references the property named `X`.

For more information about creating and using properties, see “Share Data Within App Designer Apps” on page 6-26.

Single-Source Code that Runs in Multiple Places

If you want to execute a block of code in multiple parts of your app, create a helper function. For example, you might create a helper function to update a plot after the user changes a number in an edit field or selects an item in a drop-down list. Creating a helper function allows you to single-source the common commands and avoid having to maintain redundant sets of code.

To add a helper function, expand the **Function** drop-down in the **Editor** tab, and select **Private Function** or **Public Function**. App Designer creates a template function and places your cursor in the body of that function.

To delete a helper function, select the function name in the **Functions** tab of the **Code Browser** and press the **Delete** key.

For more information about writing helper functions, see “Reuse Code Using Helper Functions” on page 6-23.

Create Input Arguments

To add input arguments to your app, click **App Input Arguments** in the **Editor** tab. Input arguments are commonly used for creating apps that have multiple windows. For more information, see “Startup Tasks and Input Arguments in App Designer” on page 6-8.

Add Help Text for Your App

Add an app summary and description to provide information about your app to users. To add help text or to edit existing help text, click **App Help Text** . Use the App Help Text dialog box to specify a short summary of the app and a more detailed explanation of what the app does and how to use it. App Designer adds this help text as a comment under the app definition statement.

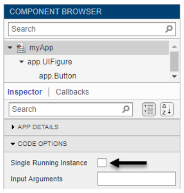
To display app help text in the MATLAB Command Window, call the `help` function and specify the app name. In addition, app help text appears at the top of the documentation page for your app. You can view the documentation page for your app by calling the `doc` function and specifying the app name.

Limit Your App to Only One Running Instance at a Time

When you create an app in App Designer you have the option to select between two run behaviors for the app:

- Allow only a single running instance of the app at a time.
- Allow multiple instances of the app to run at the same time. This is the default behavior.

To change the run behavior of your app, select the app node from the **Component Browser**. Then, from the **Code Options** section of the **App** tab, select or clear **Single Running Instance**.



Single Running Instance ☐

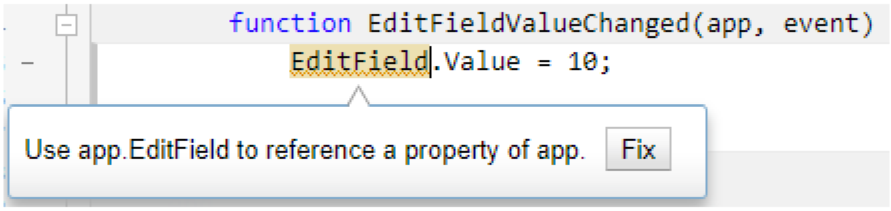
Input Arguments

When **Single Running Instance** is selected and you run the app multiple times, MATLAB reuses the existing instance and brings it to the front rather than creating a new one. When this option is cleared, MATLAB creates a new app instance each time you run it and continues to run the existing instances. These run behaviors apply to apps that you run from the **Apps** tab on the MATLAB Toolstrip or from the Command Window.

When you run apps from App Designer their behavior doesn't change whether this option is selected or cleared. App Designer always closes the existing app instance before creating a new one.

Fix Code Problems and Run-Time Errors

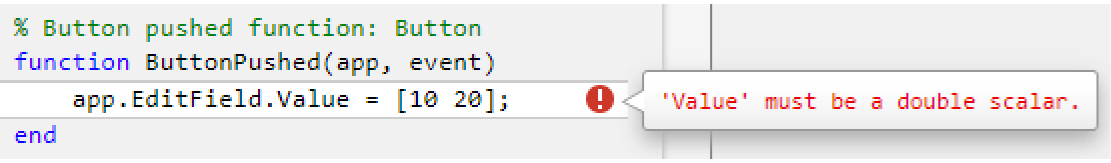
Like the MATLAB Editor, the **Code View** editor provides Code Analyzer messages to help you discover errors in your code.



```
        function EditFieldValueChanged(app, event)
            EditField.Value = 10;
```

Use app.EditField to reference a property of app.

If you run your app directly from App Designer (by clicking ▷ **Run**), App Designer highlights the source of errors in your code, should any errors occur at run time. To hide the error message, click the error indicator (the red circle). To make the error indicator disappear, fix your code and save your changes.



```
% Button pushed function: Button
function ButtonPushed(app, event)
    app.EditField.Value = [10 20];
end
```

'Value' must be a double scalar.

You can also diagnose problems in your code by debugging your app code interactively in App Designer. For more information, see “Debug MATLAB Code Files”.

Personalize Code View Appearance

You can customize how your code appears in the **Code View** editor. To change your code view preferences, go to the **Home** tab of the MATLAB Desktop. In the **Environment** section, click ⚙ **Preferences**.

Change Color Settings

To change the color settings for editable sections of code and to customize syntax highlighting, select **MATLAB** > **Colors** and adjust the desktop tool colors and the MATLAB syntax highlighting colors. These settings affect both the App Designer **Code View** editor and the MATLAB Editor. For more information, see “Change Desktop Colors”.

To change the background color of uneditable sections of code, select **MATLAB** > **App Designer** and adjust the read-only background color. This setting can be changed only if the **Use system colors** option in **MATLAB** > **Color Preferences** is unchecked.

Change Tab Preferences

To specify the size of tabs and indents in the **Code View** editor, select **MATLAB** > **Editor/Debugger** > **Tab**. From here, you can specify the size of tabs and indents, as well as details about how tabs behave. These preferences affect both the App Designer **Code View** editor and the MATLAB Editor. For more information, see “Editor/Debugger Tab Preferences”.

See Also

Related Examples

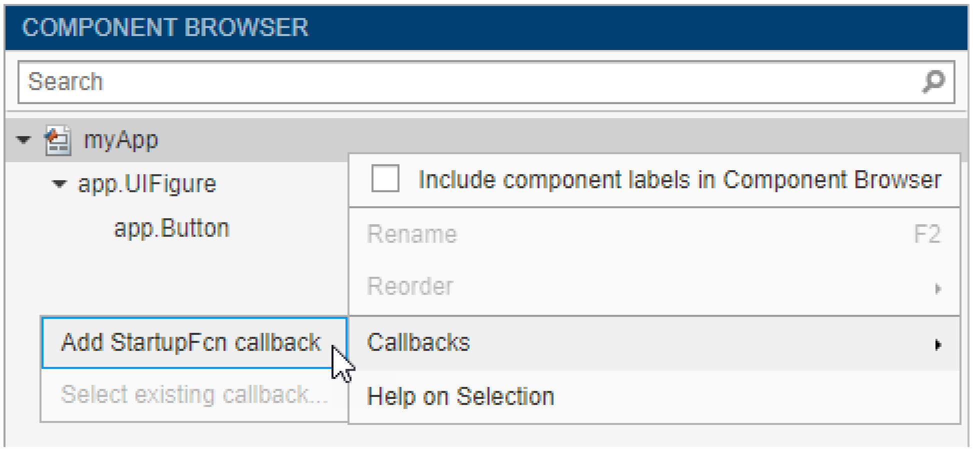
- “Callbacks in App Designer” on page 6-16
- “Share Data Within App Designer Apps” on page 6-26
- “Reuse Code Using Helper Functions” on page 6-23
- “Startup Tasks and Input Arguments in App Designer” on page 6-8

Startup Tasks and Input Arguments in App Designer

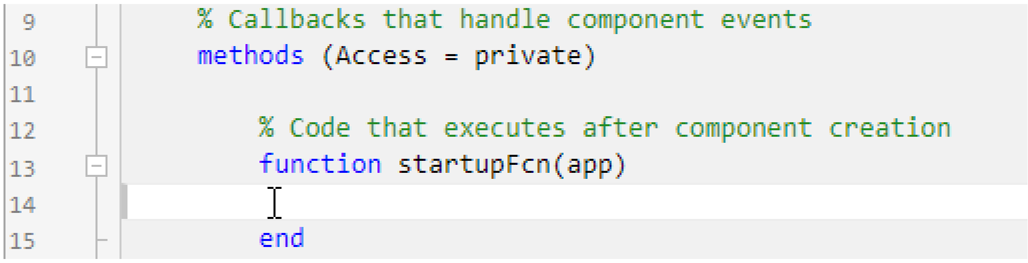
App Designer allows you to create a special function that executes when the app starts up, but before the user interacts with the UI. This function is called the `startupFcn` callback, and it is useful for setting default values, initializing variables, or executing commands that affect initial state of the app. For example, you might use the `startupFcn` callback to display a default plot or a show a list of default values in a table.

Create a startupFcn Callback

To create a `startupFcn` callback, right-click the app node from the top of the **Component Browser** hierarchy, and select **Callbacks > Add StartupFcn callback**. The app node has the same name as your MLAPP file.



App designer creates the function and places the cursor in the body of the function. Add commands to this function as you would do for any callback function. Then save and run your app.



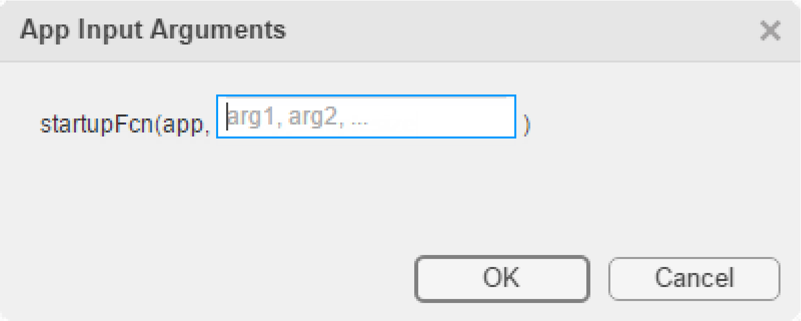
```
    % Callbacks that handle component events
    methods (Access = private)

        % Code that executes after component creation
        function startupFcn(app)

        end
```

See “App with Auto-Reflow That Updates Plot Based on User Selections” on page 7-3 for an example of an app that has a `startupFcn` callback.

Define Input App Arguments

The `startupFcn` callback is also the function where you can define input arguments for your app. Input arguments are useful for letting the user (or another app) specify initial values when the app starts up.

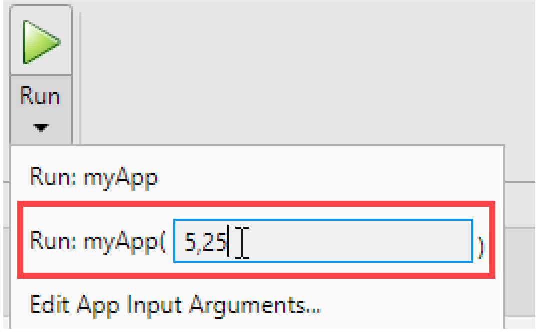
To add input arguments to an app, open the app in App Designer and click **Code View**. Then click **App Input Arguments** in the **Editor** tab.



The **App Input Arguments** dialog box allows you to add or remove input arguments in the function signature of the `startupFcn` callback. The `app` argument is always first, so you cannot change that part of the signature. Enter a comma-separated list of variable names for your input arguments. You can also enter `varargin` to make any of the arguments optional. Then click **OK**.

After you click **OK**, App Designer creates a `startupFcn` callback that has the function signature you defined in the dialog box. If your app already has a `startupFcn` callback, then the function signature is updated to include the new input arguments.

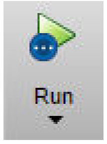
After you have created the input arguments and coded the `startupFcn`, you can test the app. Expand the drop-down list from the **Run** button in the toolstrip. In the second menu item, specify comma-separated values for each input argument. The app runs after you enter the values and press **Enter**.



Run: myApp(5,25)

Note MATLAB might return an error if you click the **Run** button without entering input arguments in the drop-down list. The error occurs because the app has required input arguments that you did not specify.

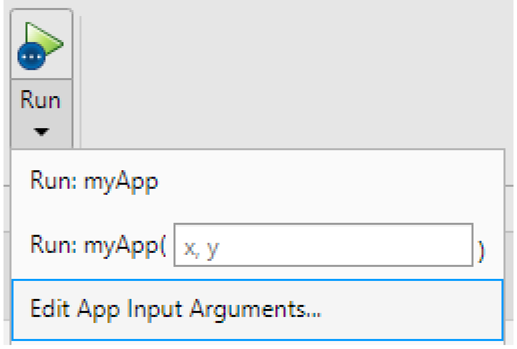
After successfully running the app with a set of input arguments, the **Run** button icon contains a blue circle.



The blue circle indicates that your last set of input values are available for re-running your app without having to type them again. Up to seven sets of input values are available to choose from.

Click the top half of the **Run** button to re-run the app with the last set of values. Or, click the bottom half of the **Run** button and select one of the previous sets of values.

The **Run** button also allows you to change the list of arguments in the function signature. Select **Edit App Input Arguments...** from the drop-down list in the bottom half of the **Run** button.



Alternatively, you can open the same **App Input Arguments** dialog box by clicking **App Input Arguments** in the toolstrip, or by right-clicking the `startupFcn` callback in the **Code Browser**.

See “Create Multiwindow Apps in App Designer” on page 6-11 for an example of an app that uses input arguments.

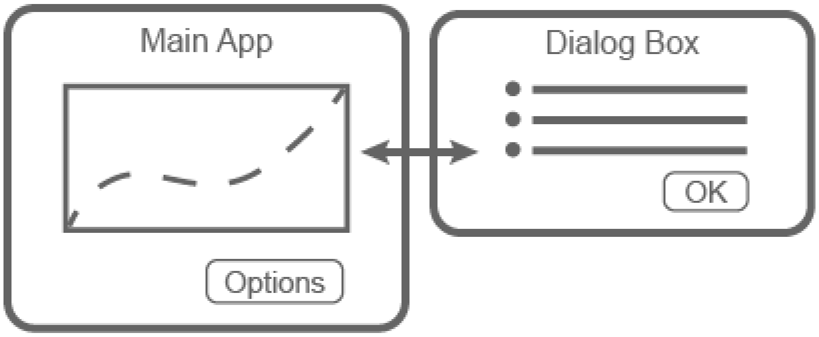
See Also

Related Examples

- “Callbacks in App Designer” on page 6-16
- “Create Multiwindow Apps in App Designer” on page 6-11
- “Add Tables to App Designer Apps” on page 4-35

Create Multiwindow Apps in App Designer

A multiwindow app consists of two or more apps that share data. The way that you share data between the apps depends on the design. One common design involves two apps: a main app and a dialog box. Typically, the main app has a button that opens the dialog box. When the user closes the dialog box, the dialog box sends the user's selections to the main window, which performs calculations and updates the UI.



These apps share information in different ways at different times:

- When the dialog box opens, the main app passes information to the dialog box by calling the dialog box app with input arguments.
- When the user clicks the **OK** button in the dialog box app, the dialog box returns information to the main app by calling a public function in the main app with input arguments.

Overview of the Process

To create the app described in the preceding section, you must create two separate apps (a main app and a dialog box app). Then perform these high-level tasks. Each task involves multiple steps.

- “Send Information to the Dialog Box” on page 6-12 — Write a `StartupFcn` callback in the dialog box app that accepts input arguments. One of the input arguments must be the main app object. Then, in the main app, call the dialog box app with the input arguments.
- “Return Information to the Main App” on page 6-13 — Write a public function in the main app that updates the UI based on the user's selections in the dialog box. Because it is a public function, the dialog box app can call it and pass values to it.
- “Manage Windows When They Close” on page 6-14 — Write `CloseRequest` callbacks in both apps that perform maintenance tasks when the windows close.

To see an implementation of all the steps in this process, see Plotting App That Opens a Dialog Box on page 6-14.

If you plan to deploy your app as a web app (requires MATLAB Compiler), creating multiple app windows is not supported. Instead, consider creating a single-window app with multiple tabs. For more information, see “Web App Limitations and Unsupported Functionality” (MATLAB Compiler).

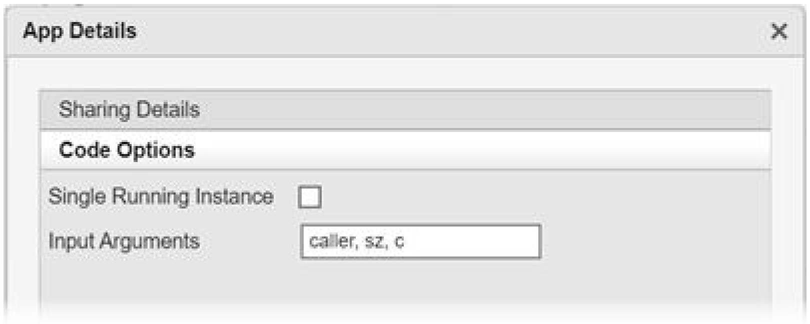
Send Information to the Dialog Box

Perform these steps to pass values from the main app to the dialog box app.

1 In the dialog box app, define input arguments for the `StartupFcn` callback function. In **Code View**, in the **Editor** tab, click **App Input Arguments** . In the App Details dialog box, enter a comma-separated list of variable names for your input arguments. Designate these inputs:

   - Main app — Pass the main app object to the dialog box app so that you can reference functions and properties of the main app from within the dialog box app code.
   - Additional data — Pass any additional data defined in the main app that the dialog box app needs access to.

   Click **OK**.



Single Running Instance ☐

Input Arguments caller, sz, c

2 In the dialog box app, add code to store the main app object.

   a First, define a property to store the main app. In **Code View**, in the **Editor** tab, select **Property** > **Private Property**. Then change the property name in the `properties` block to `CallingApp`.

```
properties (Access = private)
    CallingApp % Main app
end
```

   b Then, in the `StartupFcn` callback function, add code to store the main app object in the `CallingApp` property.

```
function StartupFcn(app,caller,sz,c)
    % Store main app object
    app.CallingApp = caller;

    % Process sz and c inputs
    % ...
end
```

   For a fully coded example of a `StartupFcn` callback, see Plotting App That Opens a Dialog Box on page 6-14.

3 In the main app, call the dialog box app from within a callback to create the dialog box.

   a First, define a property to store the dialog box app. In the main app, in **Code View**, in the **Editor** tab, select **Property** > **Private Property**. Then change the property name in the `properties` block to `DialogApp`.

```
properties (Access = private)
    DialogApp % Dialog box app
end
```

b Then, add a callback function for the **Options** button. This callback disables the **Options** button to prevent users from opening multiple dialog boxes. Next, it gets the values to pass to the dialog box, and then it calls the dialog box app with input arguments and an output argument. The output argument is the dialog box app object.

```
function OptionsButtonPushed(app,event)
    % Disable Plot Options button while dialog is open
    app.OptionsButton.Enable = "off";

    % Get sample size and colormap
    % ...

    % Call dialog box with input values
    app.DialogApp = DialogAppExample(app,szvalue,cvalue);
end
```

For a fully coded example of a callback, see Plotting App That Opens a Dialog Box on page 6-14.

Return Information to the Main App

Perform these steps to return the user's selections from the dialog box app to the main app.

1. In the main app, create a public function that updates the UI. With the main app open in **Code View**, in the **Editor** tab, select **Function > Public Function**.

   Change the default function name to the desired name, and add input arguments for each option you want to pass from the dialog box to the main app. The `app` argument, which represents the main app object, must be first, so specify the additional arguments after that argument. Then add code to the function that processes the inputs and updates the main app.

   ```
   function updateplot(app,sz,c)
       % Process sz and c
       ...
   end
   ```

   For a fully coded example of a public function, see Plotting App That Opens a Dialog Box on page 6-14.

2. In the dialog box app, call the public function from within a callback. With the dialog box app open in **Code View**, add a callback function for the **OK** button.

   In this callback, call the public function that you defined in the main app code. Pass the main app object, stored in the `CallingApp` property, as the first argument. Then, pass the additional data that the main app needs to update its UI. Finally, call the `delete` function to close the dialog box.

   ```
   function ButtonPushed(app,event)
      % Call main app's public function
      updateplot(app.CallingApp,app.EditField.Value,app.DropDown.Value);

      % Delete the dialog box
      delete(app)
   end
   ```

Manage Windows When They Close

Both apps must perform certain tasks when the user closes them. Before the dialog box closes, it must re-enable the **Options** button in the main app. Before the main app closes, it must ensure that the dialog box is closed.

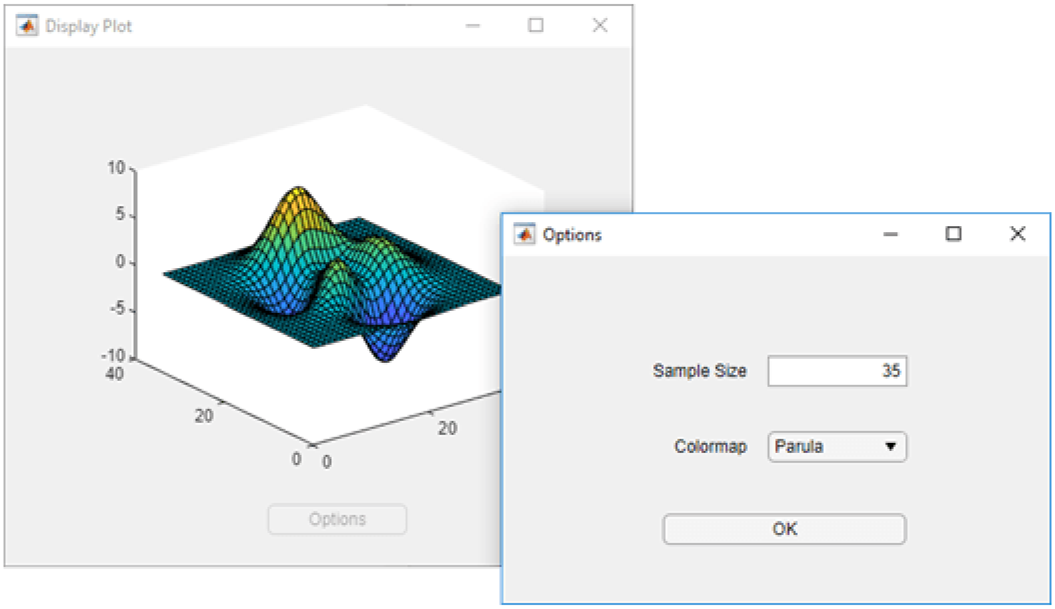
1. With the dialog box app open in **Code View**, right-click the `app.UIFigure` object in the **Component Browser** and select **Callbacks > Add CloseRequestFcn callback**. Then add code that re-enables the button in the main app and closes the dialog box app.

   ```
   function DialogAppCloseRequest(app,event)
      % Enable the Plot Options button in main app
      app.CallingApp.OptionsButton.Enable = "on";

      % Delete the dialog box
      delete(app)
   end
   ```
2. With the main app open in **Code View**, right-click the `app.UIFigure` object in the **Component Browser** and select **Callbacks > Add CloseRequestFcn callback**. Then add code that closes both apps.

   ```
   function MainAppCloseRequest(app,event)
      % Delete both apps
      delete(app.DialogApp)
      delete(app)
   end
   ```

Example: Plotting App That Opens a Dialog Box

This app consists of a main plotting app that has a button for selecting options in a dialog box. The **Options** button calls the dialog box app with input arguments. In the dialog box, the callback for the **OK** button sends the user's selections back to the main app by calling a public function in the main app.



Sample Size 35

Colormap Parula

See Also

More About

- “Callbacks in App Designer” on page 6-16
- “Startup Tasks and Input Arguments in App Designer” on page 6-8

Callbacks in App Designer

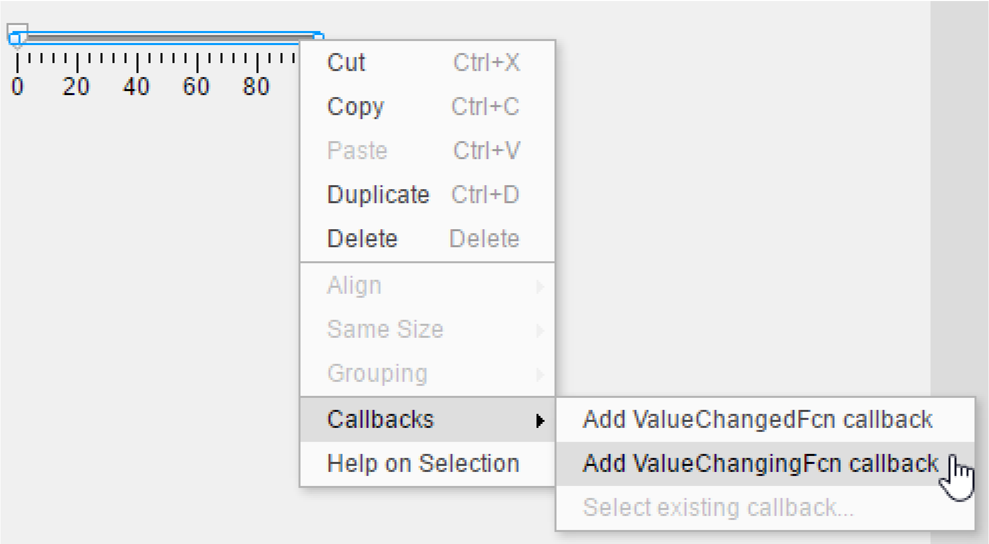
A callback is a function that executes when a user interacts with a UI component in your app. You can use callbacks to program the behavior of your app. For example, you can write a callback that plots some data when an app user clicks a button, or a callback that moves the needle of a gauge component when a user interacts with a slider.

Most components have at least one callback, and each callback is tied to a specific interaction with the component. However, some components, such as labels and lamps, do not have callbacks because those components only display information. To see the list of callbacks that a component supports, select the component and click the **Callbacks** tab in the **Component Browser**.

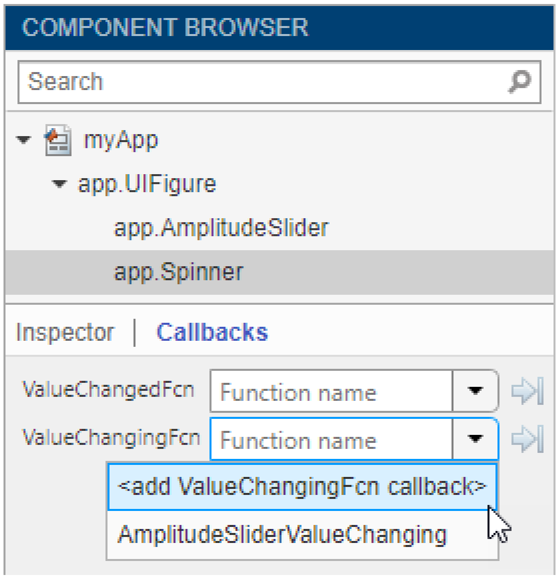
Create Callback Functions

There are several ways to create a callback for a UI component. You can take different approaches depending on where you are working in App Designer. Choose the most convenient approach from this list:

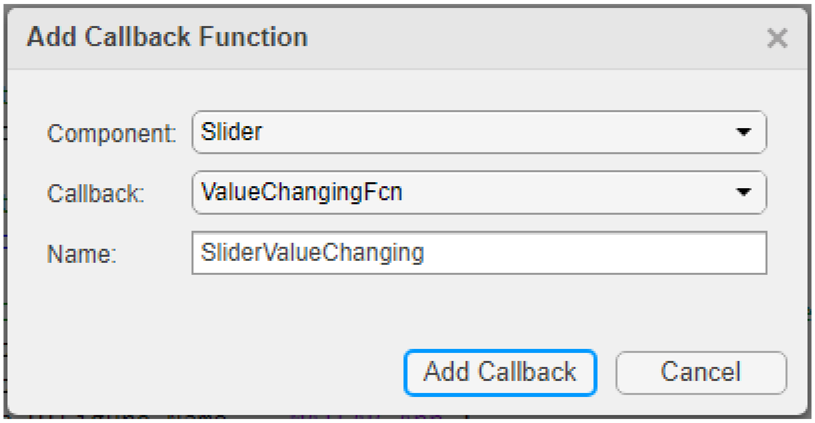
- Right-click a component in the canvas, **Component Browser**, or **App Layout** pane, and select **Callbacks > Add (callback property) callback**.



- Select the **Callbacks** tab in the **Component Browser**. The left side of the **Callbacks** tab shows the supported callback properties. The drop-down list next to each callback property allows you to specify a name for the callback function or to select a default name in angle brackets <>. If your app has existing callbacks, the drop-down list includes those callbacks. Select an existing callback when you want multiple UI components to execute the same code.



- In code **Code View**, in the **Editor** tab, click **Callback**. Alternatively, in the **Code Browser** pane, on the **Callbacks** tab, click the ✚ button.



Component: Slider

Callback: ValueChangingFcn

Name: SliderValueChanging

Specify these options in the Add Callback Function dialog box:

- **Component** — Specify the UI component that executes the callback.
- **Callback** — Specify the callback property. The callback property maps the callback function to a specific interaction. Some components have more than one callback property available. For example, sliders have two callback properties: `ValueChangedFcn` and `ValueChangingFcn`. The `ValueChangedFcn` callback executes after the user moves the slider and releases the mouse. The `ValueChangingFcn` callback for the same component executes repeatedly while the user moves the slider.
- **Name** — Specify a name for the callback function. App Designer provides a default name, but you can change it in the text field. If your app has existing callbacks, the **Name** field has a drop-down arrow next to it, indicating that you can select an existing callback from a list.

Program Callback Functions

When you create a callback for a component, App Designer generates a callback function in **Code View** and places your cursor in the function. Write code in this callback function to program the callback behavior.

Callback Input Arguments

All callback functions that App Designer creates have these input arguments in the function signature:

- `app` — The `app` object. Use this object to access UI components in the app as well as other variables stored as properties.
- `event` — An object that contains specific information about the app user's interaction with the UI component.

The `app` argument provides the `app` object to your callback. You can access any component (and all component-specific properties) within any callback by using this syntax:

```
app.Component.Property
```

For example, this command sets the `Value` property of a gauge to `50`. In this case, the name of the gauge is `PressureGauge`:

```
app.PressureGauge.Value = 50;
```

The `event` argument provides an object that has different properties, depending on the specific callback that is executing. The object properties contain information that is relevant to the type of interaction that the callback is responding to. For example, the `event` argument in a `ValueChangingFcn` callback of a slider contains a property called `Value`. That property stores the slider value as the user moves the thumb (before the user releases the mouse). Here is a slider callback function that uses the `event` argument to make a gauge track the value of the slider:

```
function SliderValueChanging(app,event)
      latestvalue = event.Value; % Current slider value
      app.PressureGauge.Value = latestvalue;  % Update gauge
end
```

To learn more about the `event` argument for a specific component's callback function, see the property page for that component. Right-click the component, and select **Help on Selection** to open the property page. For a list of property pages for all UI components, see “App Building Components” on page 4-2.

Share Data Between Callback Functions

To store data that needs to be accessed by multiple callbacks, create a property. Properties contain data that belongs to the app. You can create private properties to store data to be shared within the app only, or public properties to store data to be shared outside of the app (for example, with a script, function, or other app that needs access to the data).

Create a public or private property by clicking the **Property** button in the **Editor** tab in **Code View**. Enter a name for your property. You can then assign and access the property value within all of your app callbacks using the syntax `app.`*`PropertyName`*.

For more information, see “Share Data Within App Designer Apps” on page 6-26.

Share Callbacks Between Multiple Components

Sharing callbacks between components is useful when you want to offer multiple ways of doing something in your app. For example, your app can respond the same way when a user clicks a button or presses the **Enter** key in an edit field.

You can create a single shared callback for multiple selected components with a callback type in common. For example, in an app with an edit field and a slider, you can select both components, right-click one of them, and select **Callbacks > Add ValueChangingFcn callback**. App Designer creates a single new callback and assigns it to both the edit field and the slider.

Alternatively, after you create a callback for one component, you can share it by assigning it to a second component. Right-click the second component in the **Component Browser** and select **Callbacks** > **Select existing callback**. When the Select Callback Function dialog box displays, select the existing callback from the **Name** drop-down list.

For an example of an app that shares a callback between two components, see “Use One Callback for Multiple App Designer Components” on page 6-31.

Create and Assign Callbacks Programmatically

You can also create and assign callback functions programmatically in your app code. Use this method to create a callback for a component or graphics object that does not appear in the **Component Browser**. For example, you can programmatically assign a callback to a dialog box that you create in your app code, or to a `Line` object that you plot in a `UIAxes` component.

Create the callback function as a private function by selecting **Function** > **Private Function** in the **Editor** tab of the toolstrip. The function must have `app`, `src`, and `event` as the first three arguments. Here is an example of a callback written as a private function:

```
methods (Access = private)

        function myclosefcn(app,src,event)
            disp('Have a nice day!');
        end

end
```

Assign the callback function to a component by specifying the callback property value as a handle to your callback function using the syntax `@app.`*`FunctionName`*. For example, this code creates an alert dialog box that assigns the `myclosefcn` function to the `CloseFcn` callback property. The function executes when the dialog box closes.

```
uialert(app.UIFigure,"File not found","Alert", ...
    "CloseFcn",@app.myclosefcn);
```

To write a callback function that accepts additional input arguments, specify the additional arguments after the first three arguments. For example, this callback accepts one additional input, `name`:

```
methods (Access = private)

        function displaymsg(app,src,event,name)
            msg = name + " dialog box closed";
            disp(msg);
        end
end
```

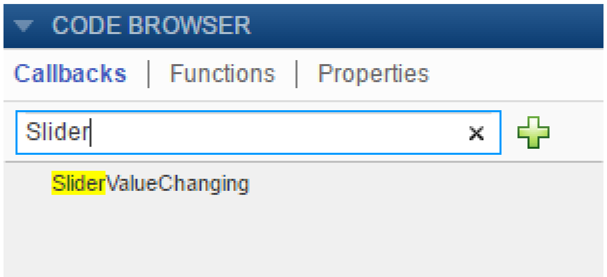
To assign this callback to a component, specify the component callback property as cell array. The first element in the cell array must be the function handle. Subsequent elements must be the additional input values. For example:

```
uialert(app.UIFigure,"File not found","Alert", ...
    "CloseFcn",{@app.displaymsg,"Alert"});
```

For more information, see “Add UI Components to App Designer Programmatically” on page 4-20.

Search for Callbacks in Your Code

If your app has a lot of callbacks, you can quickly search and navigate to a specific callback by typing part of the name in the search bar at the top of the **Callbacks** tab in the **Code Browser**. After you begin typing, the **Callbacks** pane clears, except for the callbacks that match your search.



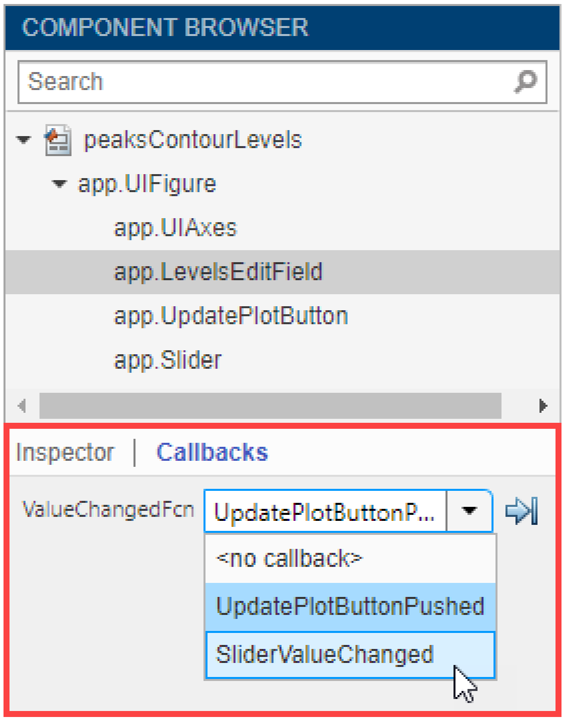
Slider

SliderValueChanging

Click a search result to scroll the callback into view. Right-clicking a search result and selecting **Go To** places your cursor in the callback function.

Change or Disconnect Callbacks

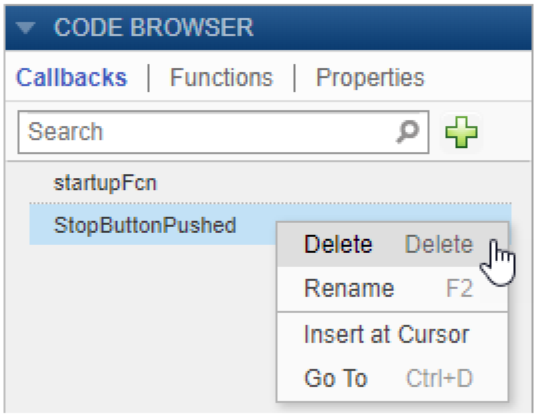
To assign a different callback to a component, select the component in the **Component Browser**. Then click the **Callbacks** tab and select a different callback from the drop-down list. The drop-down list displays only the existing callbacks.



To disconnect a callback that is shared with a component, select the component in the **Component Browser**. Then click the **Callbacks** tab and select **<no callback>** from the drop-down menu. Selecting this option only disconnects the callback from the component. It does not delete the function definition from your code, nor does it disconnect the callback from any other components. After you disconnect a callback, you can create a new callback for the component or leave the component without a callback function.

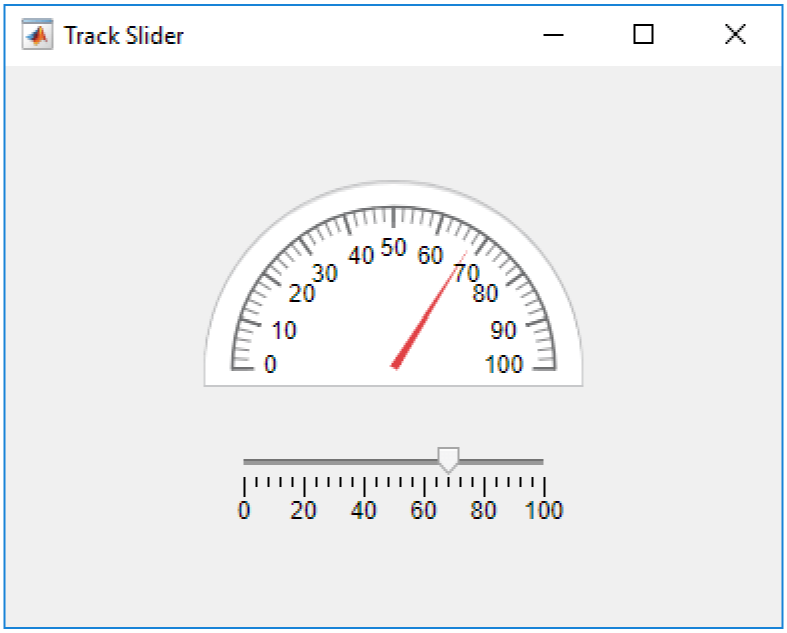
Delete Callbacks

If your code contains a callback function that is not being used by any components in your app, you can delete the function entirely. Delete a callback by right-clicking the callback in the **Callbacks** tab of the **Code Browser** and selecting **Delete** from the context menu.



Example: App with a Slider Callback

This app contains a gauge that tracks the value of a slider as the user moves the thumb. The `ValueChangingFcn` callback for the slider gets the current value of the slider from the `event` argument. Then it moves the gauge needle to that value.



See Also

Related Examples

- “Share Data Within App Designer Apps” on page 6-26
- “Use One Callback for Multiple App Designer Components” on page 6-31
- “Add UI Components to App Designer Programmatically” on page 4-20
- “Create Callbacks for Apps Created Programmatically” on page 11-2

Reuse Code Using Helper Functions

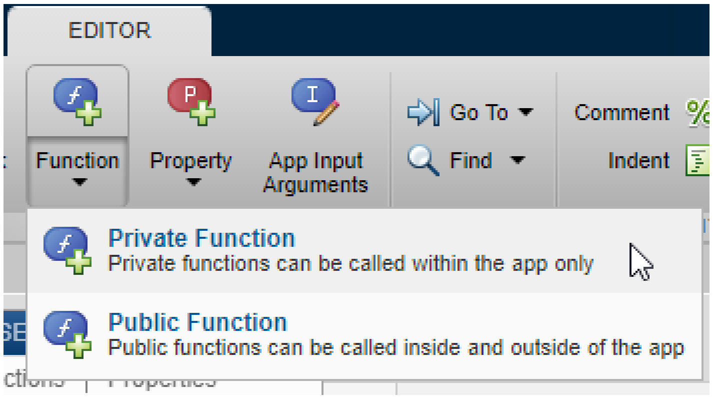
Helper functions are MATLAB functions that you define in your app so that you can call them at different places in your code. For example, you might want to update a plot after the user changes a number in an edit field or selects an item in a drop-down list. Creating a helper function allows you to single-source the common commands and avoid having to maintain redundant code.

There are two types of helper functions: private functions, which you can call only inside your app, and public functions, which you can call either inside or outside your app. Private functions are commonly used in single-window apps, while public functions are commonly used in multiwindow apps.

Create a Helper Function

Code View provides a few different ways to create a helper function:

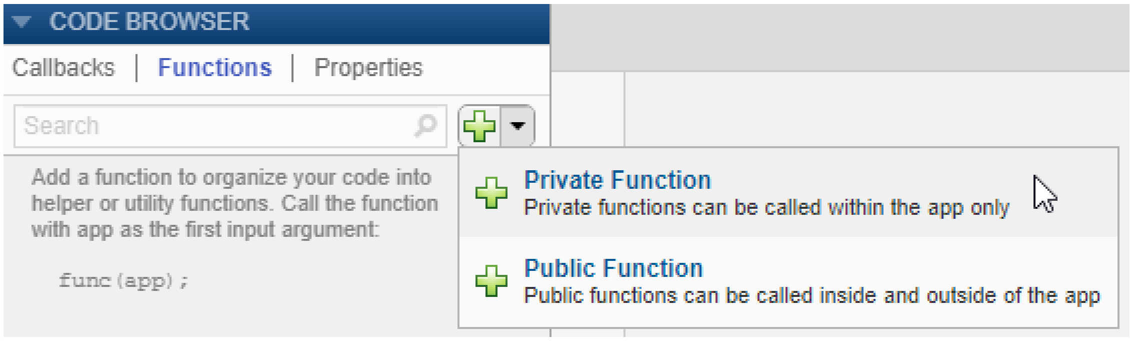
- Expand the drop-down list from the bottom half of the **Function** button in the **Editor** tab. Select **Private Function** or **Public Function**.



Private functions can be called within the app only

Public functions can be called inside and outside of the app

- Select the **Functions** tab in the **Code Browser**, expand the drop-down list on the ✚▾ button, and select **Private Function** or **Public Function**.



Add a function to organize your code into helper or utility functions. Call the function with app as the first input argument:

```
func(app);
```

Private functions can be called within the app only

Public functions can be called inside and outside of the app

When you make your selection, App Designer creates a template function and places your cursor in the body of that function. Then you can update the function name and its arguments, and add your code to the function body. The `app` argument is required, but you can add more arguments after the `app` argument. For example, this function creates a surface plot of the `peaks` function. It accepts an additional argument `n` for specifying the number of samples to display in the plot.

```
methods (Access = private)

        function updateplot(app,n)
            surf(app.UIAxes,peaks(n));
            colormap(app.UIAxes,winter);
        end

end
```

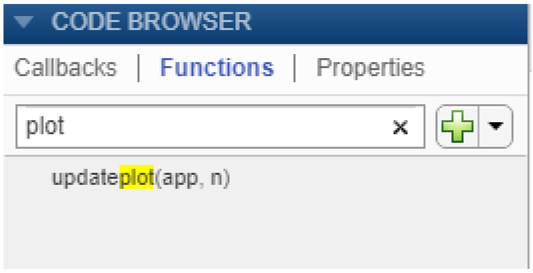
Call the function from within any callback. For example, this code calls the `updateplot` function and specifies `50` as the value for `n`.

```
updateplot(app,50);
```

Managing Helper Functions

Managing helper functions in the **Code Browser** is similar to managing callbacks. You can change the name of a helper function by double-clicking the name in the **Functions** tab of the **Code Browser** and typing a new name. App Designer automatically updates all references to the function when you change its name.

If your app has numerous helper functions, you can quickly search and navigate to a specific function by typing part of the name in the search bar at the top of the **Functions** tab. After you begin typing, the **Functions** tab clears, except for the items that match your search.

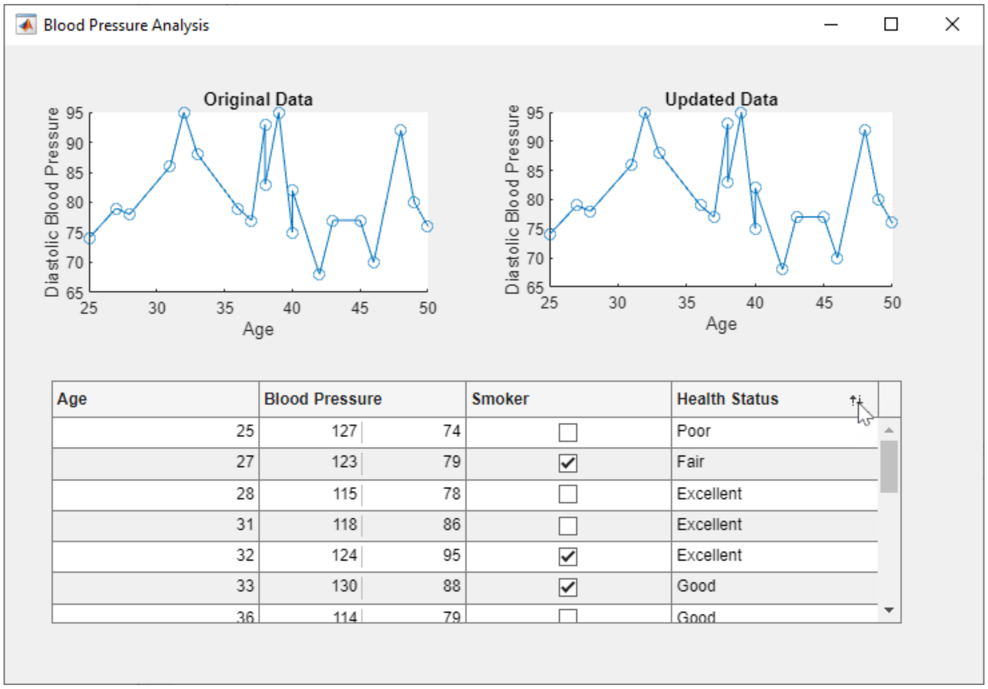


Click a search result to scroll the function into view. Right-clicking a search result and selecting **Go To** places your cursor in the function.

To delete a helper function, select its name in the **Functions** tab and press the **Delete** key.

Example: Helper Function That Initializes Plots and Displays Updated Data

This app shows how to create a helper function that initializes two plots and updates one of them in a component callback. The app calls the `updateplot` function at the end of the `StartupFcn` callback when the app starts up. The `UITableDisplayDataChanged` callback calls the same function to update one of the plots when the user sorts columns or changes a value in the table.



Age	Blood Pressure		Smoker	Health Status
25	127	74	□	Poor
27	123	79	☑	Fair
28	115	78	□	Excellent
31	118	86	□	Excellent
32	124	95	☑	Excellent
33	130	88	☑	Good
36	114	79	□	Good

See Also

Related Examples

- "Callbacks in App Designer" on page 6-16
- "Create Multiwindow Apps in App Designer" on page 6-11

Share Data Within App Designer Apps

Using properties is the best way to share data within an app because properties are accessible to all functions and callbacks in an app. All UI components are properties, so you can use this syntax to access and update UI components within your callbacks:

```
app.Component.Property
```

For example, these commands get and set the `Value` property of a gauge. In this case, the name of the gauge is `PressureGauge`.

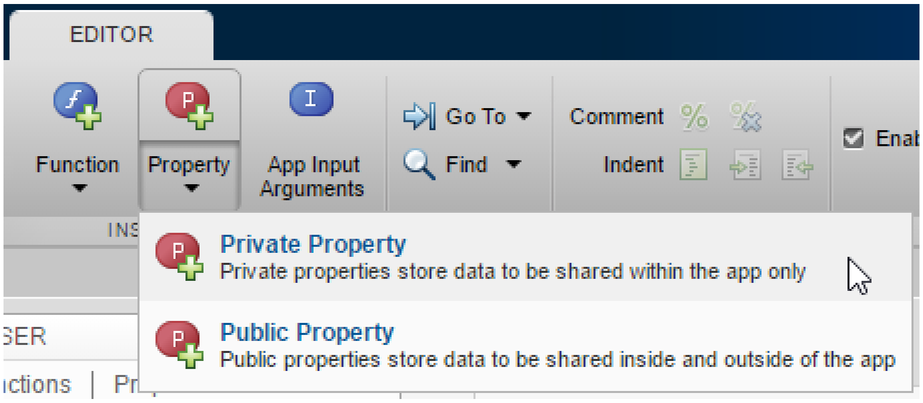
```
x = app.PressureGauge.Value; % Get the gauge value
app.PressureGauge.Value = 50; % Set the gauge value to 50
```

However, if you want to share an intermediate result, or data that multiple callbacks need to access, then define a public or private property to store your data. Public properties are accessible both inside and outside of the app, whereas private properties are only accessible inside of the app.

Define a Property

Code View provides a few different ways to create a property:

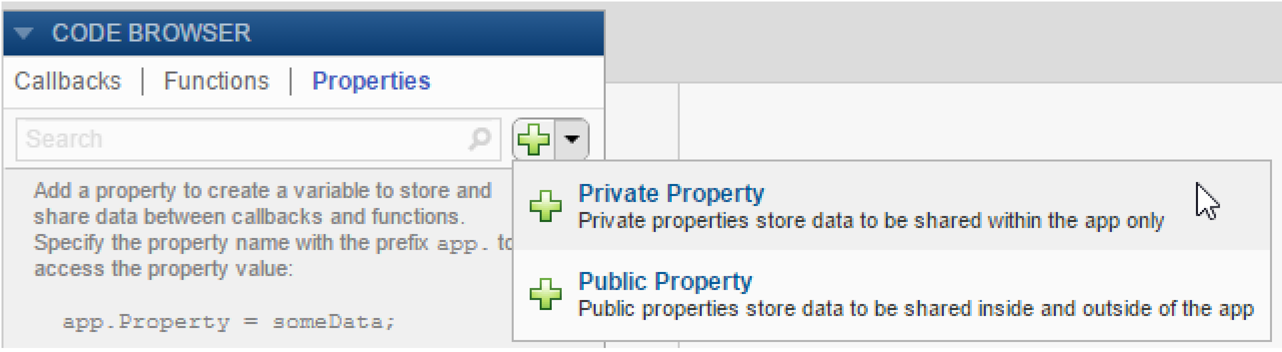
- Expand the drop-down menu from the bottom half of the **Property** button in the **Editor** tab. Select **Private Property** or **Public Property**.



Private properties store data to be shared within the app only

Public properties store data to be shared inside and outside of the app

- Click on the **Properties** tab in the **Code Browser**, expand the drop-down list on the ✚▾ button, and select **Private Property** or **Public Property**.



Add a property to create a variable to store and share data between callbacks and functions. Specify the property name with the prefix app. to access the property value:

```
app.Property = someData;
```

Private properties store data to be shared within the app only

Public properties store data to be shared inside and outside of the app

After you select an option to create a property, App Designer adds a property definition and a comment to a `properties` block.

```
properties (Access = public)
    Property % Description
end
```

The `properties` block is editable, so you can change the name of the property and edit the comment to describe the property. For example, this property stores a value for average cost:

```
properties (Access = public)
    X % Average cost
end
```

If your code needs to access a property value when the app starts, you can initialize its value in the `properties` block or in the `startupFcn` callback.

```
properties (Access = public)
    X = 5; % Average cost
end
```

To restrict the types of values that a property can store, associate a data type with the property in the property definition. For example, this code requires that values assigned to `X` must be of a type that is compatible with `double`, and any assigned values are stored as a `double`.

```
properties (Access = public)
    X double % Average cost
end
```

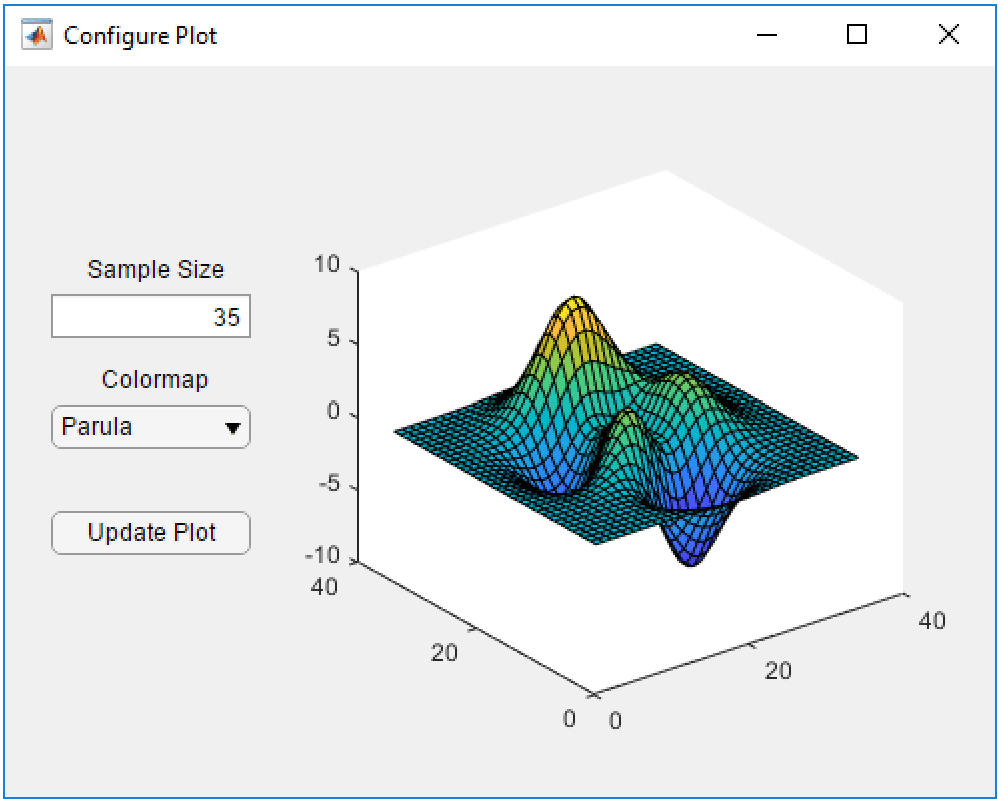
Access a Property

Once you define a property, you can access and set the property value anywhere in your app code by using the syntax `app.`*`PropertyName`*.

```
y = app.X  % Get the value of X
app.X = 5; % Set the value of X
```

Example: Share Plot Data and a Drop-Down List Selection

This app shows how to share data in a private property and a drop-down list. It has a private property called `Z` that stores plot data. The callback function for the edit field updates `Z` when the user changes the sample size. The callback function for the **Update Plot** button gets the value of `Z` and the colormap selection to update the plot.



See Also

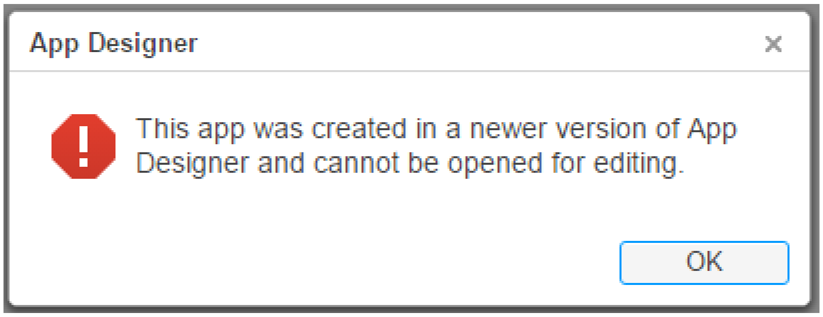
Related Examples

- "Callbacks in App Designer" on page 6-16
- "Create Multiwindow Apps in App Designer" on page 6-11

Compatibility Between Different Releases of App Designer

Starting in R2018a, the apps you save in App Designer have a new format. This new file format might impact your ability to edit newer apps in previous releases, but it has no impact on your ability to run them in previous releases.

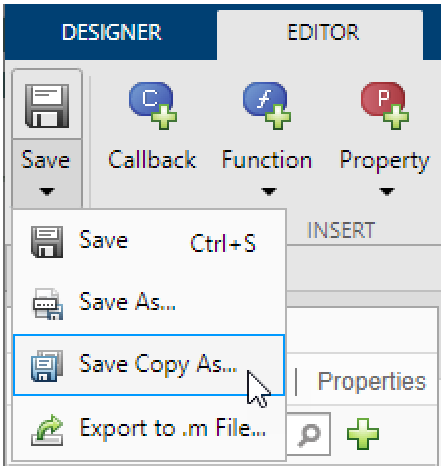
If you try to edit an app, created in R2018a or later, in an earlier release of App Designer, the new format is not recognized after saving your changes. You see a message such as this.



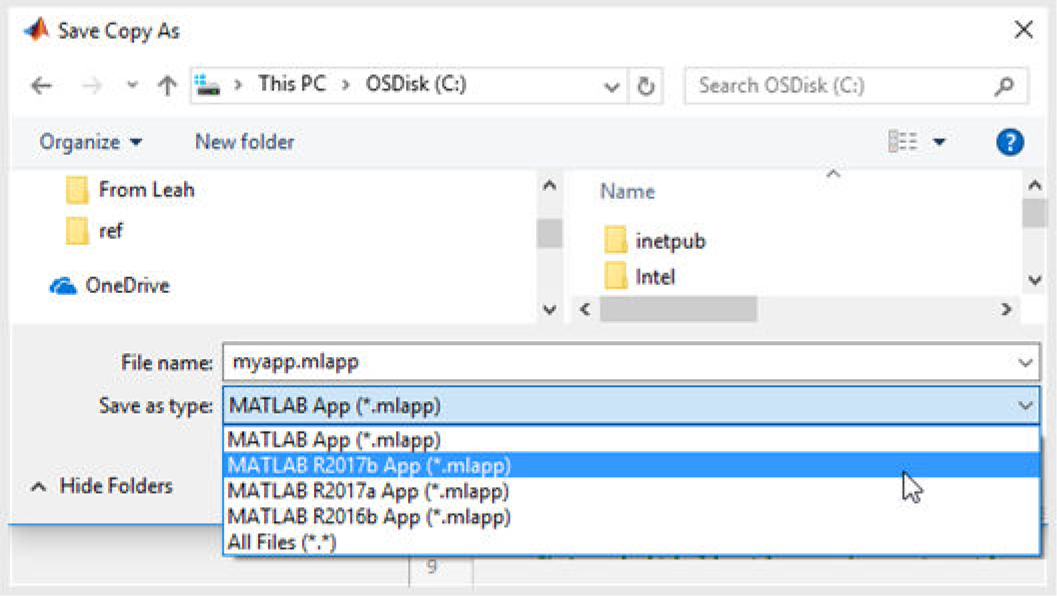
App Designer

This app was created in a newer version of App Designer and cannot be opened for editing.

To enable editing of newer apps in a previous release, save the app in the release-specific format. Select **Save** > **Save Copy As** from any of the tabs in the toolstrip.



In the Save Copy As dialog box, select a type from the **Save as type** drop-down list.



File name: myapp.mlapp

Save as type: MATLAB App (*.mlapp)

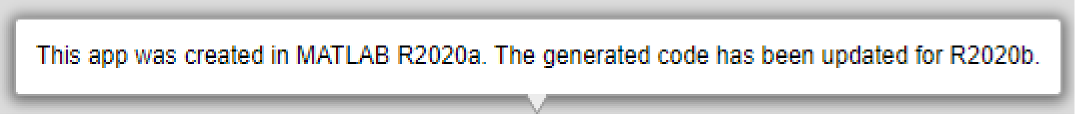
Save Copy As Versus Save As

The **Save Copy As** and **Save As** options serve different purposes, and their behavior is also different.

- To save your app in a format that can be edited in earlier releases, use **Save Copy As**. When you use this option, App Designer saves the copy of the app in the specified folder, but it does not replace the app in your current session.
- To save a copy of your app that is editable only with the current release, use **Save As**. When you use this option, App Designer saves the copy of the app in the specified folder and replaces the app in your current session.

Opening Apps for Editing in a Newer Release

If you open an app for editing that was created in a previous release, App Designer updates the app and displays a message such as this one.



This app was created in MATLAB R2020a. The generated code has been updated for R2020b.

See Also

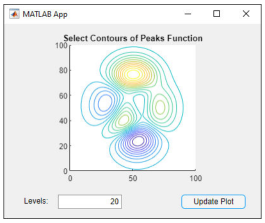
App Designer

Use One Callback for Multiple App Designer Components

Sharing callbacks between components is useful when you want to offer multiple ways of doing something in your app. For example, your app can respond the same way when a user clicks a button or presses the **Enter** key in an edit field.

Example of a Shared Callback

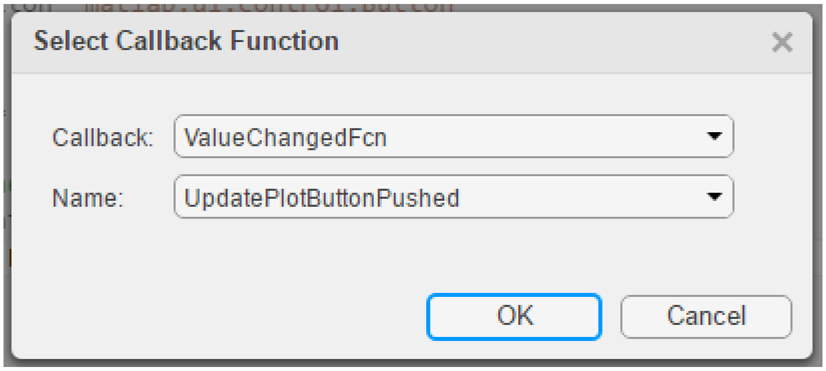
This example shows how to create an app containing two UI components that share a callback. The app displays a contour plot with the specified number of levels. When the user changes the value in the edit field, they can press **Enter** or click the **Update Plot** button to update the plot.



Levels: 20

1. In App Designer, drag an **Axes** component from the **Component Library** onto the canvas. Then make these changes:
   - Double-click the title, and change it to `Select Contours of Peaks Function`.
   - Double-click the X and Y axis labels, and press the **Delete** key to remove them.
2. Drag an **Edit Field (Numeric)** component below the axes on the canvas. Then make these changes:
   - Double-click the label next to the edit field and change it to `Levels:`.
   - Double-click the edit field and change the default value to `20`.
3. Drag a **Button** component next to the edit field on the canvas. Then double-click its label and change it to `Update Plot`.
4. Add a callback function that executes when the user clicks the button. Right-click the **Update Plot** button and select **Callbacks** > **Add ButtonPushedFcn callback**.
5. App Designer switches to the **Code View**. Paste this code into the body of the `UpdatePlotButtonPushed` callback:

   ```
   Z = peaks(100);
   nlevels = app.LevelsEditField.Value;
   contour(app.UIAxes,Z,nlevels);
   ```
6. Next, share the callback with the edit field. In the **Component Browser**, right-click the `app.LevelsEditField` component and select **Callbacks** > **Select existing callback**. When the Select Callback Function dialog box displays, select **UpdatePlotButtonPushed** from the **Name** drop-down list.



Select Callback Function

Callback: ValueChangedFcn

Name: UpdatePlotButtonPushed

Sharing this callback allows the user to update the plot after changing the value in the edit field and pressing **Enter**. Alternatively, they can change the value and press the **Update Plot** button.

7. Next, set the axes aspect ratio and limits. In the **Component Browser**, select the `app.UIAxes` component. Then, make the following changes in the **Axes** tab:

   - Set **PlotBoxAspectRatio** to `1,1,1`.
   - Set **XLim** and **YLim** to `0,100`.

8. Click **Run** to save and run the app.

See Also

Related Examples

- “Callbacks in App Designer” on page 6-16
- “Share Data Within App Designer Apps” on page 6-26
- “Manage Code in App Designer Code View” on page 6-2

7

App Designer Examples

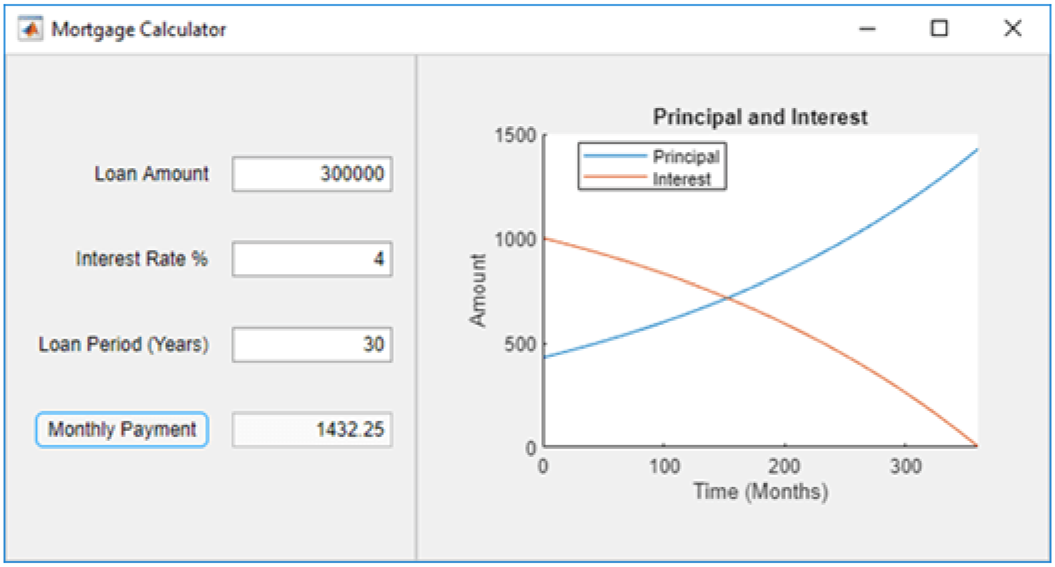
App That Calculates and Plots Data Based on Numerical Input

This app shows how to use numeric edit fields to create a simple mortgage amortization calculator. It includes the following components to collect user input, calculate monthly payments, and plot the principal and interest amounts over time:

- Numeric edit fields — allow users to enter values for the loan amount, interest rate, and loan period. MATLAB® automatically checks to make sure the values are numeric and within the range specified by the app. A fourth numeric edit field displays the resulting monthly payment amount based on the inputs.
- Push button — executes a callback function to calculate the monthly payment value.
- Axes — used to plot the principal and interest amounts versus mortgage installment.

To open the app in App Designer, enter this command in the MATLAB Command Window:

```
openExample('matlab/MortgageCalculatorExample')
```



Mortgage Calculator

Loan Amount 300000

Interest Rate % 4

Loan Period (Years) 30

Monthly Payment 1432.25

See Also

UIAxes

Related Examples

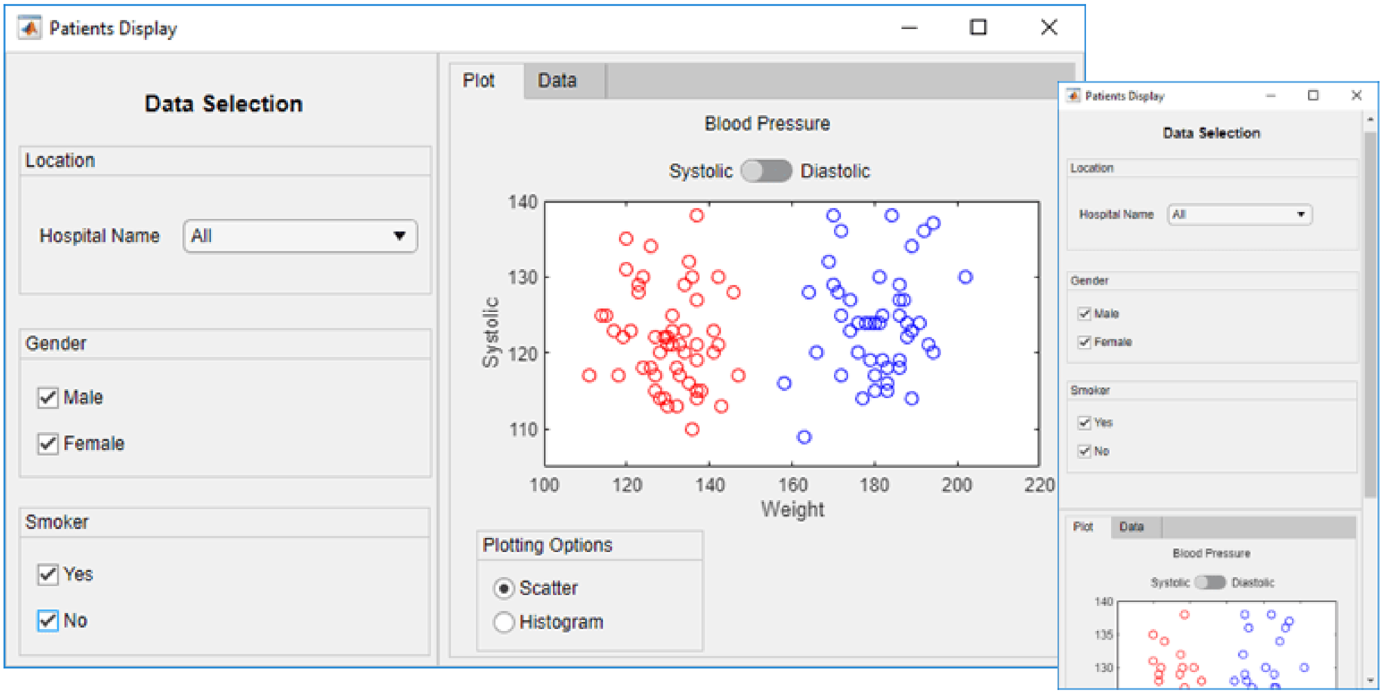
- “Callbacks in App Designer” on page 6-16

App with Auto-Reflow That Updates Plot Based on User Selections

This app shows how to define controls and tabs within the panels of an app with auto-reflow. The controls are in an anchored panel on the left. The right panel that reflows contains two tabs. One tab displays a chart and user interface components for adjusting the chart. The other tab contains a table with the data used to make the chart. User selections update both the plot and the table. The app responds to resizing by automatically growing, shrinking, and reflowing the app content.

The app includes these components:

- Check boxes — used to update the plot and table when the user selects or clears a check box.
- Switch — used to toggle the data that is visualized in the chart
- Button group containing radio buttons — used to manage exclusive selection of radio buttons. When the user selects a radio button, the button group executes a callback function to update the plot with the appropriate data.
- Slider — used to adjust histogram bin width. This slider only appears when the **Histogram** plotting option is selected in the button group.
- Table — used to view the data associated with the chart.



Hospital Name: All

☑ Male
☑ Female

☑ Yes
☑ No

● Scatter
○ Histogram

See Also

UIAxes | Table

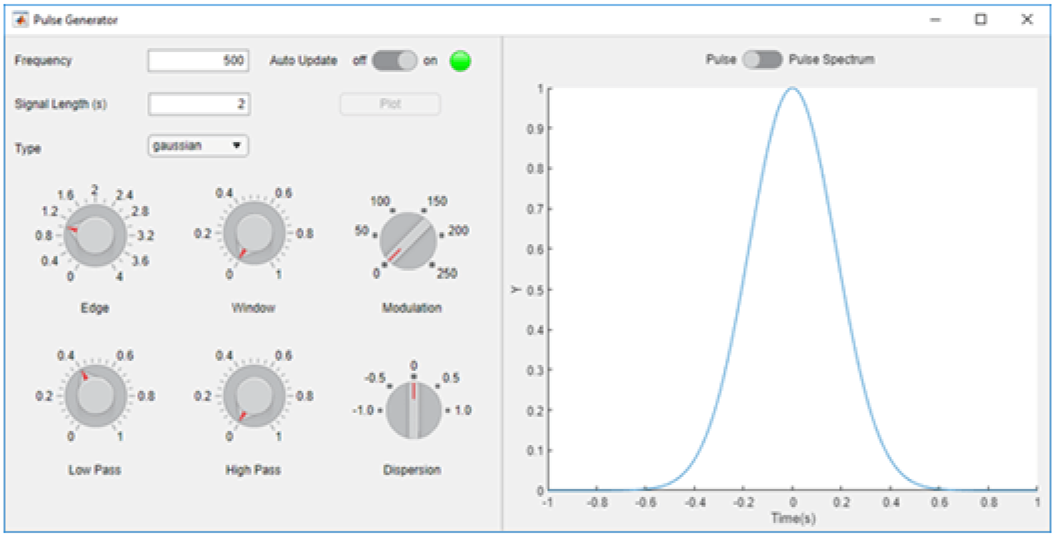
Related Examples

- “Callbacks in App Designer” on page 6-16

App That Uses Grid Layout to Manage Component Positions and Resizing

This app shows how to use a grid layout manager to control the alignment and resizing of knobs when the app is resized. The app also uses the following components to gather user input and plot the resulting wave form:

- Numeric edit fields — allow users to enter the pulse frequency and length. MATLAB® automatically checks to make sure the values are numeric and within the range specified by the app.
- Switches — allow users to control automatic plot updates and toggle between plots in the time and frequency domains.
- Drop-down menu — allows users to select from a list of pulse shapes, such as Gaussian, sinc, and square.
- Knobs — allow users to modify the pulse by specifying a window function, modulating the signal, or applying other enhancements.



Pulse Generator

Frequency 500

Signal Length (s) 2

Type gaussian

See Also

Functions
`uigridlayout`

Properties
UIAxes

Related Examples

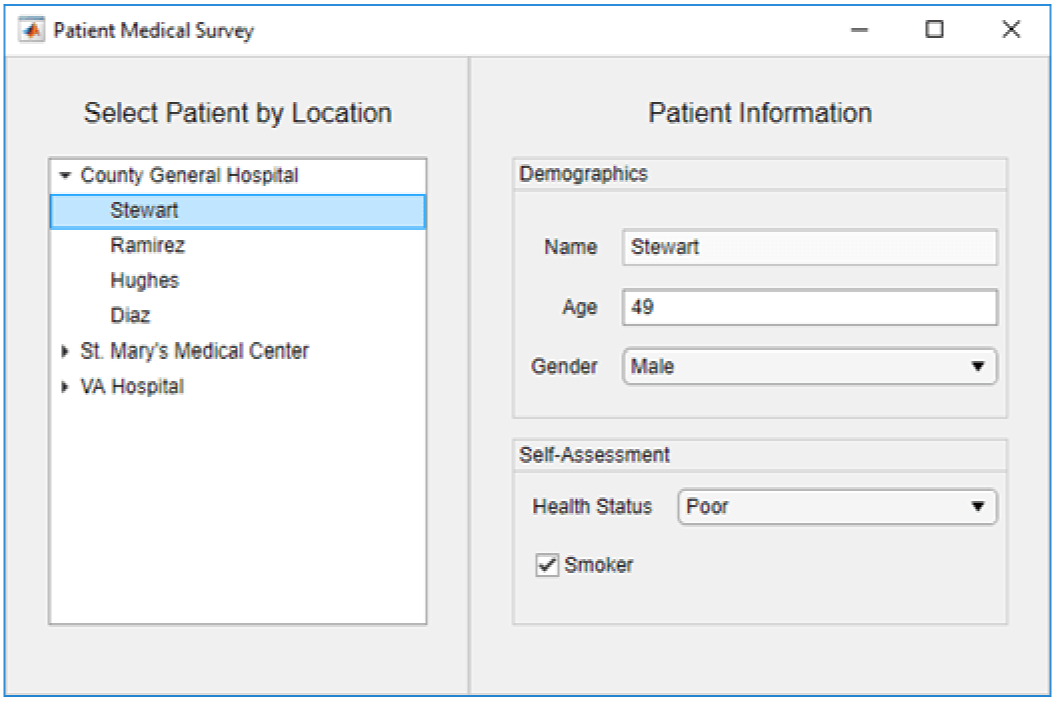
- “Callbacks in App Designer” on page 6-16

App That Displays Data in a Hierarchy Using Tree

This app shows how to add a tree to an App Designer app. The app selects data from `patients.xls` and displays it in a hierarchy using a tree. The tree contains three nodes that display hospital names. Each hospital node contains nodes that display patient names. When the user clicks a patient name in the tree, the **Patient Information** panel displays data such as age, gender, and health status. When the user edits the patient data, the app asks the user to confirm the change and then stores the change in the table variable.

In addition to the tree and **Patient Information** panel, this app also contains the following UI components:

- Read-only text field — Used to display the patient's name
- Numeric edit field — Used to display and accept changes to the patient's age
- Drop-down list — Used to display and accept changes to the patient's gender and health status
- Check box — Used to display and accept changes to the patient's smoking history
- Confirmation dialog box — Used to confirm changes to patient data



County General Hospital
Stewart
Ramirez
Hughes
Diaz
St. Mary's Medical Center
VA Hospital

Name: Stewart
Age: 49
Gender: Male
Health Status: Poor
☑ Smoker

See Also

`uitree` | `uitreenode` | `readtable` | `table`

Related Examples

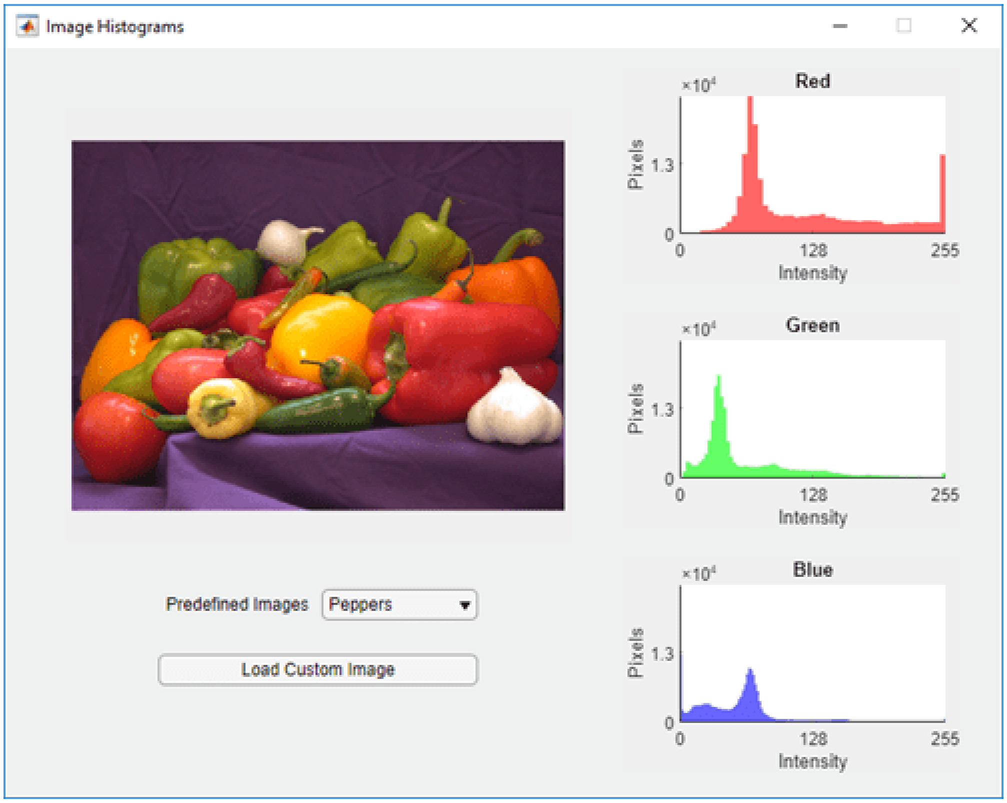
- "Add UI Components to App Designer Programmatically" on page 4-20

Create App That Uses Multiple Axes to Display Results of Image Analysis

This app shows how to configure multiple axes components in App Designer. The app displays an image in one axes component, and displays histograms of the red, green, and blue pixels in the other three.

This example also demonstrates the following app building tasks:

- Managing multiple axes
- Reading and displaying images
- Browsing the user’s file system using the `uigetfile` function
- Displaying an in-app alert for invalid input (in this case, an unsupported image file)
- Writing a `StartupFcn` callback to initialize the app with a default image



See Also

Functions
`imagesc` | `imread` | `uialert`

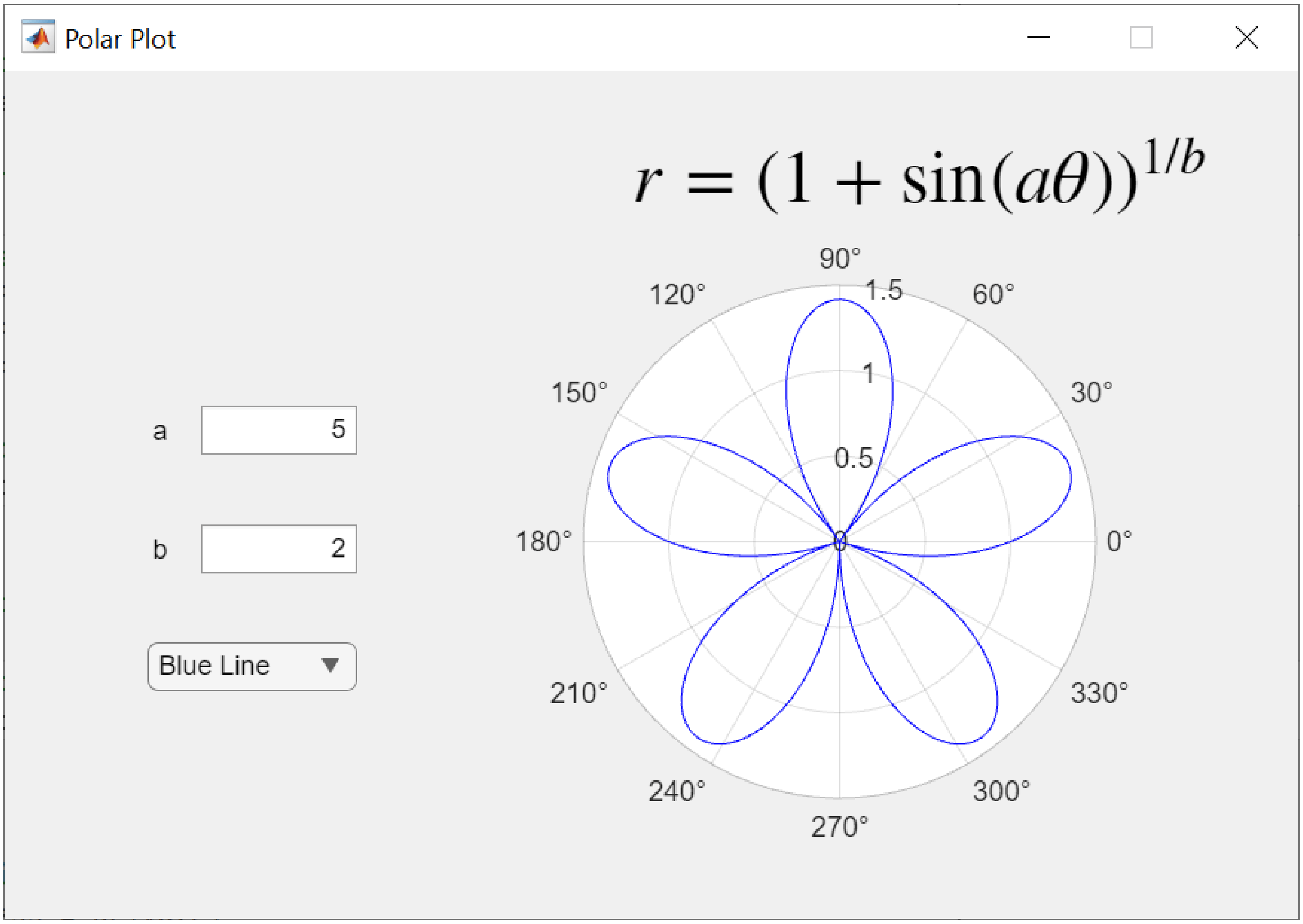
Properties
UIAxes

Create Polar Axes Programmatically in an App

This app shows how to display a plot by creating the axes programmatically before calling a plotting function. In this case, the app plots a polar equation using the `polaraxes` and `polarplot` functions. When the user changes the value of *a* or *b*, or when they select a different line color, the plot updates to reflect their changes.

This example also demonstrates these app building concepts:

- Creating different types of axes programmatically to display plots that `uiaxes` does not support
- Calling a plotting function in App Designer
- Sharing a callback with multiple components
- Displaying an equation in a label using LaTeX formatting



See Also

`polaraxes` | `polarplot`

Related Examples

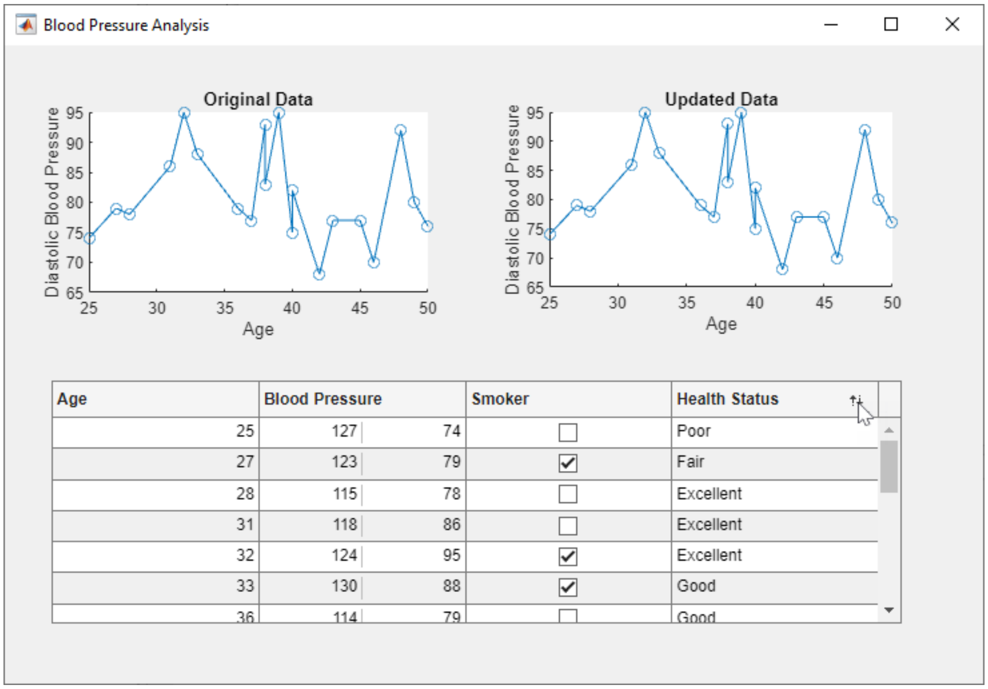
- “Display Graphics in App Designer” on page 3-15

Create App with a Table That Can Be Sorted and Edited Interactively

This app shows how to display data in a table UI component. The app loads a spreadsheet into a table array when the app starts up. Then it displays and plots a subset of the data from the spreadsheet. One of the plots updates when the user edits values or sorts columns in the table UI component at run time.

This example demonstrates the following app building tasks:

- Displaying the contents of a table array in a table UI component
- Enabling some of the interactive features of a table UI component



Age	Blood Pressure		Smoker	Health Status
25	127	74	□	Poor
27	123	79	☑	Fair
28	115	78	□	Excellent
31	118	86	□	Excellent
32	124	95	☑	Excellent
33	130	88	☑	Good
36	114	79	□	Good

See Also

`readtable` | `table` | `uitable`

Related Examples

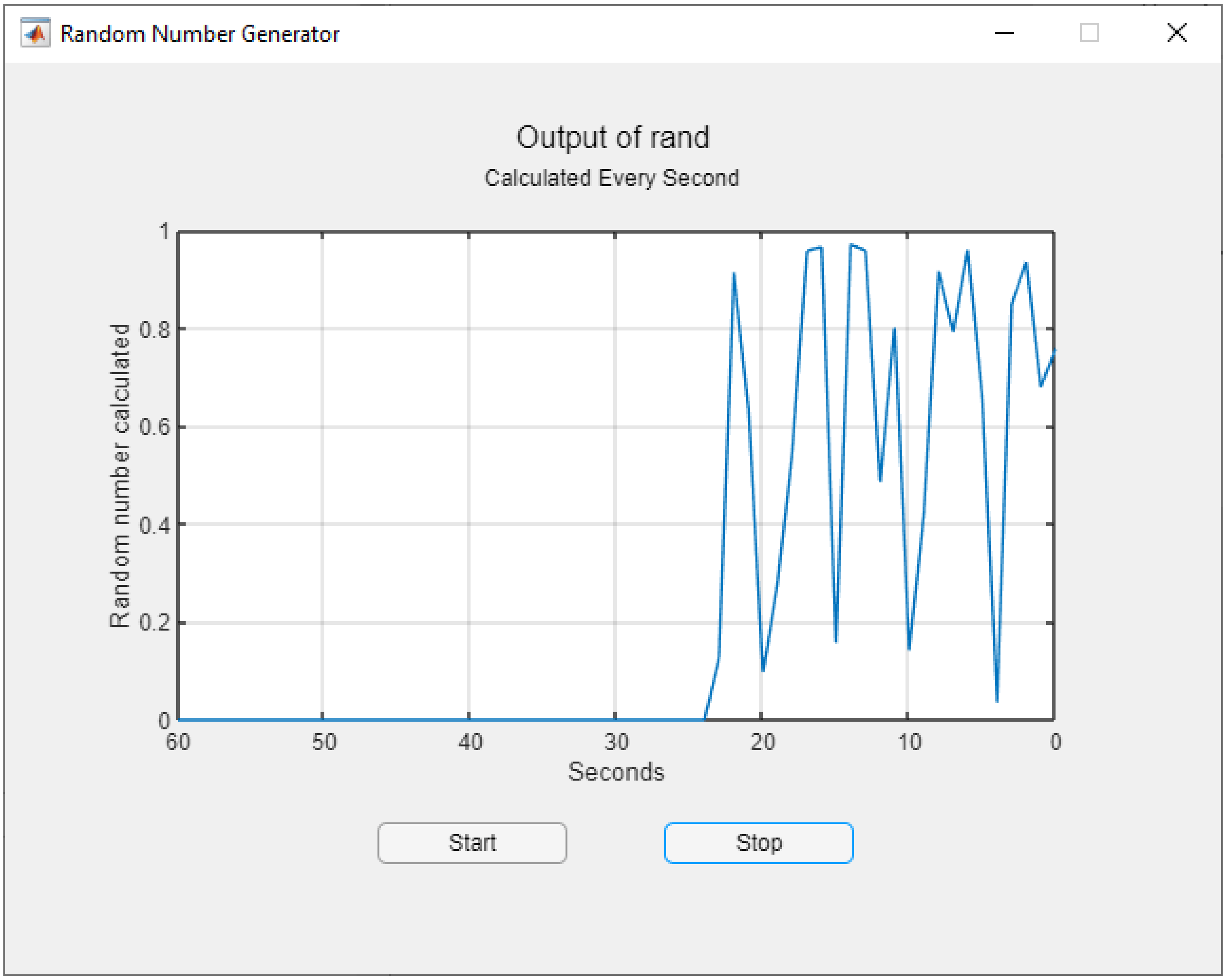
- “Display Tabular Data in Apps” on page 4-15
- “Programmatic App That Displays a Table” on page 15-8

Create App with Timer Object Configured Programmatically

This app shows how to create a timer object in App Designer that executes a function at regular time intervals. In this case, the app generates random data every second and plots the result.

This example also demonstrates the following app building tasks:

- Writing a callback for an object created programmatically (in this case, the timer object)
- Configuring a timer object to execute its callback at regular intervals
- Starting the timer when the user clicks the **Start** button
- Stopping the timer when the user clicks the **Stop** button
- Deleting the timer when the app closes



See Also

Functions
`timer` | `memory`

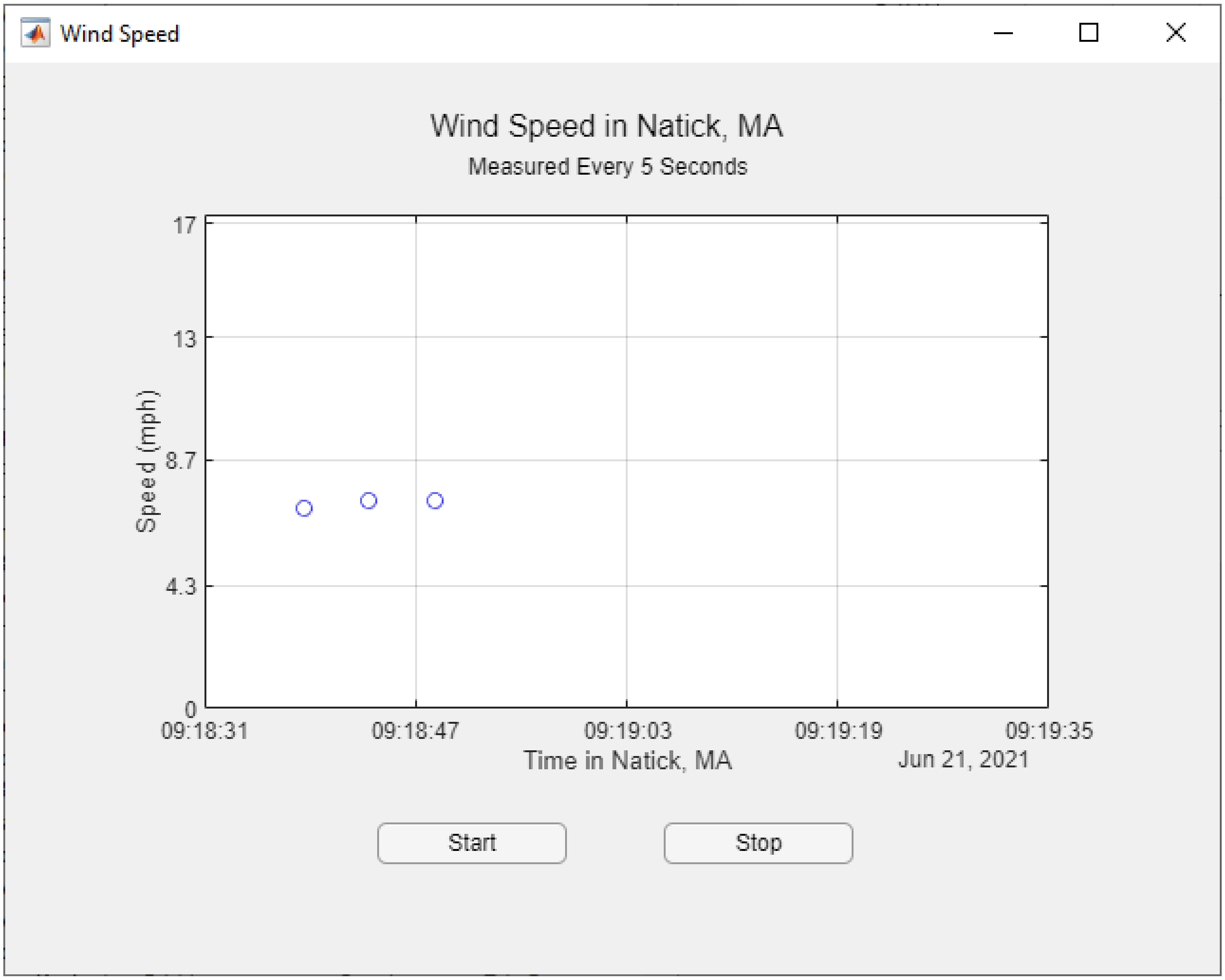
Properties
UIAxes

Create App with Timer Object That Queries Website Data

This app shows how to create a timer object in App Designer that executes a function at regular time intervals. In this case, the app queries the wind speed from a web site every five seconds and plots the returned value.

This example also demonstrates the following app building tasks:

- Writing a callback for an object created programmatically (in this case, the timer object)
- Configuring a timer object to execute its callback at regular intervals
- Starting the timer when the user clicks the **Start** button
- Stopping the timer when the user clicks the **Stop** button
- Deleting the timer when the app closes



See Also

Classes
`timer`

Functions
`webread`

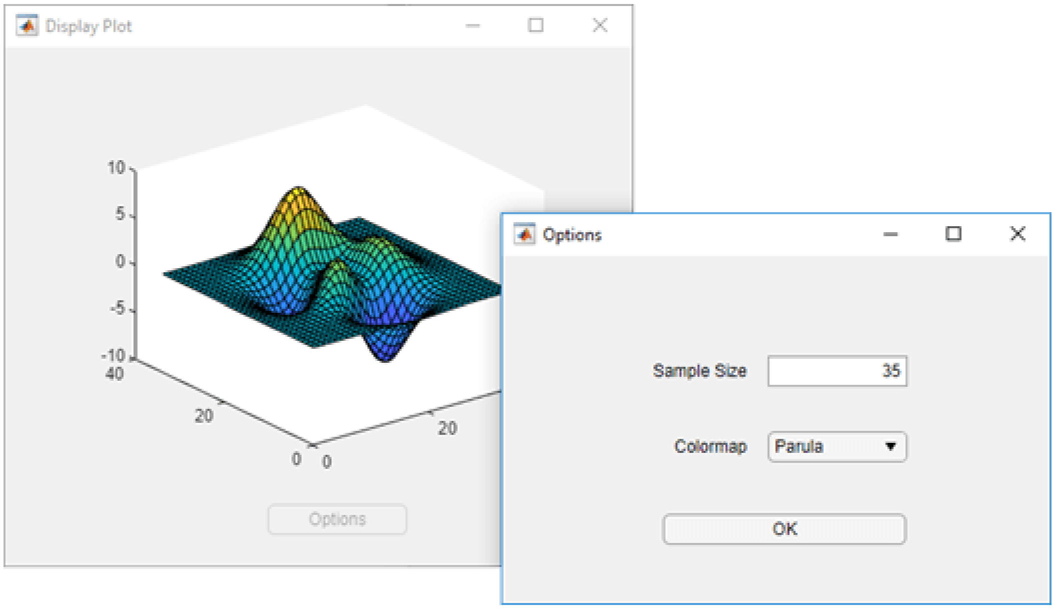
Properties
UIAxes

Share Data in Multiwindow Apps

This example shows how to pass data from one app to another. This multiwindow app consists of a main app that calls a dialog box app with input arguments. The dialog box displays a set of options for modifying aspects of the main app. When the user closes it, the dialog box sends their selections back to the main app.

This example demonstrates the following app building tasks:

- Calling an app with input arguments
- Calling an app with a return argument that is the app object
- Passing values to an app by calling a public function in the app
- Writing `CloseRequestFcn` callbacks to perform maintenance tasks when each app closes



Sample Size 35

Colormap Parula

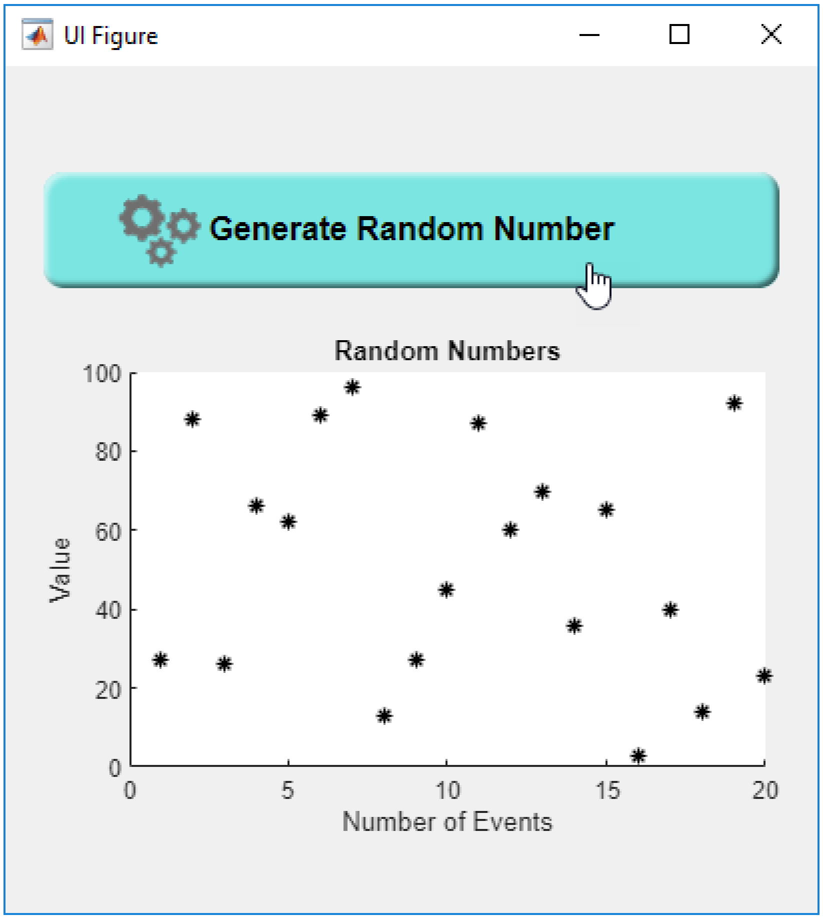
See Also

Related Examples

- “Create Multiwindow Apps in App Designer” on page 6-11
- “Startup Tasks and Input Arguments in App Designer” on page 6-8
- “Reuse Code Using Helper Functions” on page 6-23

Display HTML Elements Styled by a Cascading Style Sheet

This app shows how to reference supporting files from your HTML file, like a Cascading Style Sheet and an image used by the CSS file. This app also demonstrates how to plot data in MATLAB® that is generated in JavaScript® when an HTML button is clicked.



See Also

Functions
`uihtml`

Properties
HTML Properties

More About

- "Create HTML Content in Apps" on page 4-23

Advanced App Designer Examples

Organize App Data Using MATLAB Classes

In this section...

As the size and complexity of an app increases, it can be difficult to organize and manage the code to perform calculations, process data, and manage user interactions in one file. This example shows how to take an app created entirely in App Designer and reorganize the app code into two parts:

- Code that stores your app data and the algorithms to process that data, implemented as a MATLAB class
- Code that displays the app and manages user interactions, implemented as an App Designer app

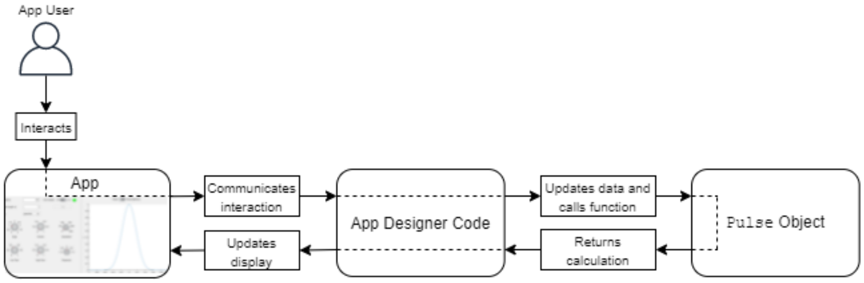
Separating the data and algorithms from the app has multiple benefits.

- **Scalability** — It is easier to extend app functionality when the code is organized into multiple self-contained portions.
- **Reusability** — You can reuse your data and algorithms across multiple apps with minimal effort.
- **Testability** — You can run and test your algorithms in MATLAB, independently from the app.

This example uses the `PulseGenerator` app, which lets users specify options to generate a pulse and visualize the resulting waveform. The goal of the example is to reorganize the code in the original app by performing these steps:

1. Create a `Pulse` class that stores pulse data, such as the type, frequency, and length of the pulse, and the algorithm used to take that pulse data and generate the resulting waveform.
2. Modify the code in App Designer to use the `Pulse` class to perform calculations and to update the app display.

In the final app, when a user interacts with the app controls, the code in App Designer updates the data stored in the `Pulse` class and calls a class method to generate the waveform data. App Designer then updates the app display with the new waveform visualization.



To view and run the final app, see “Pulse Generator App That Stores Data in a Class” on page 8-6.

Open App Designer App

Run this command to open a working copy of the PulseGenerator app.

```
openExample('matlab/PulseGeneratorAppExample')
```

Use this app as a starting point as you modify and reorganize the app code.

Write a MATLAB Class to Manage App Data

Separating the data and algorithms that are independent of the app interface allows you to organize the different tasks that your code performs, and to test and reuse these tasks independently of one another. Implementing this portion of your app as a MATLAB class has these benefits:

- You can manage a large amount of interdependent data using object-oriented design.
- You can easily share and update this data within your App Designer app.

For more information about the benefits of object-oriented design in MATLAB, see “Why Use Object-Oriented Design”.

Define Class

To determine which aspects of your app to separate out as a class, consider what parts of your app code do not directly impact the app user interface, and which parts of your app you might want to test separately from the running app.

In the pulse generator app, the app data consists of the pulse that the user wants to visualize. Create a new class file named Pulse.m in the same folder as the PulseGenerator.mlapp app file. Define a handle class named Pulse by creating a classdef block.

```
classdef Pulse < handle
% ...
end
```

Store your app data and write functions to implement your app algorithms within the classdef block.

Create Properties

Use properties to store and share app data. To define properties, create a properties block. Create properties for data that the app needs access to and for data that is processed by algorithms associated with the app.

In the Pulse class, create a properties block to hold the data that defines a pulse, such as the pulse type and the frequency and length of the pulse.

```
    properties
        Type
        Frequency
        Length
        Edge
        Window
        Modulation
        LowPass
        HighPass
        Dispersion
    end

    properties (Constant)
        StartFrequency = 10;
        StopFrequency = 20;
    end
```

For more information about defining properties in a class, see “Property Syntax”.

Create Functions

Define functions that operate on the app data in a `methods` block in the class definition.

For example, the original `PulseGenerator` app has a function defined in App Designer named `generatePulse` that computes a pulse based on the pulse properties. Because this algorithm does not need to update the app display or directly respond to user interaction, you can move the function definition from App Designer into the `Pulse` class.

Create a `methods` block and copy the `generatePulse` function definition into the block. To keep the class definition independent of the app, update the references to UI component values in the app to instead query the values of `Pulse` object properties using the syntax `obj.`*`Property`*. The beginning of your function definition should look like this:

```
methods
    function result = generatePulse(obj)

        type = obj.Type;
        frequency = obj.Frequency;
        signalLength = obj.Length;
        edge = obj.Edge;
        window = obj.Window;
        modulation = obj.Modulation;
        lowpass = obj.LowPass;
        highpass = obj.HighPass;
        dispersion = obj.Dispersion;

        startFrequency = obj.StartFrequency;
        stopFrequency = obj.StopFrequency;

        t = -signalLength/2:1/frequency:signalLength/2;
        sig = (signalLength/(8*edge))^2;

        switch type
           % The rest of the code is the same as the original
           % function in the PulseGenerator app.
           % ...
    end
end
```

To view the complete function code, see “Pulse Generator App That Stores Data in a Class” on page 8-6.

For more information about writing class methods, see “Method Syntax”.

Test Algorithm

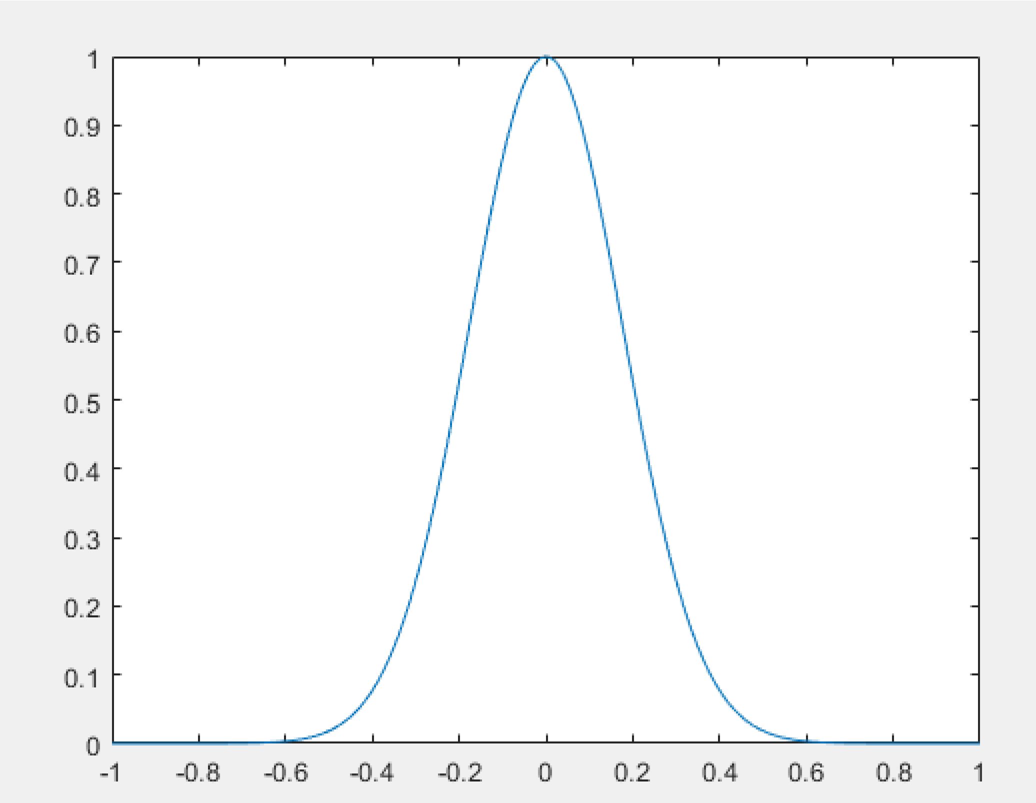
One of the benefits of storing app data in a class is that you can interact with the data object and test your algorithms independently the running app.

For example, create a `Pulse` object and set its properties in the Command Window.

```
p = Pulse;
p.Type = 'gaussian';
p.Frequency = 500;
p.Length = 2;
p.Edge = 1;
p.Window = 0;
p.Modulation = 0;
p.LowPass = 0.4;
p.HighPass = 0;
p.Dispersion = 0;
```

Call the `generatePulse` method of the `Pulse` object `p`. Visualize the pulse in a plot.

```
step = 1/p.Frequency;
xlim = p.Length/2;
x = -xlim:step:xlim;
y = generatePulse(p);
plot(x,y);
```



You can also test your algorithm using a testing framework. For more information, see “Ways to Write Unit Tests”.

Share Data with App

To access the data object from within App Designer, create an instance of the class in your App Designer code and store it in a property of your app. You can set and query the object properties that store the data and call the class functions to process the data in response to user interactions.

In the `PulseGenerator` app in App Designer, create a new private property by clicking the **Property** button in the **Editor** tab. Add a private property named `PulseObject` to hold the `Pulse` object.

Then, in the `StartupFcn` for the app, create a `Pulse` object by adding this code to the top of the function definition.

```
app.PulseObject = Pulse;
```

To generate the pulse for visualization when a user interacts with one of the controls in the app, modify the `updatePlot` function. This function is called in multiple callback functions of the `PulseGenerator` app, whenever the user interacts with one of the controls in the app.

In the `updatePlot` function, first set the properties of the `app.Pulse` object using the values of the app controls by adding this code to the top of the function.

```
app.PulseObject.Type = app.TypeDropDown.Value;
app.PulseObject.Frequency = app.FrequencyEditField.Value;
app.PulseObject.Length = app.SignalLengthsEditField.Value;
app.PulseObject.Edge = app.EdgeKnob.Value;
app.PulseObject.Window = app.WindowKnob.Value;
app.PulseObject.Modulation = str2double(app.ModulationKnob.Value);
app.PulseObject.LowPass = app.LowPassKnob.Value;
app.PulseObject.HighPass = app.HighPassKnob.Value;
app.PulseObject.Dispersion = str2double(app.DispersionKnob.Value);
```

Then, update the call to the `generatePulse` function by replacing the input argument with `app.PulseObject`.

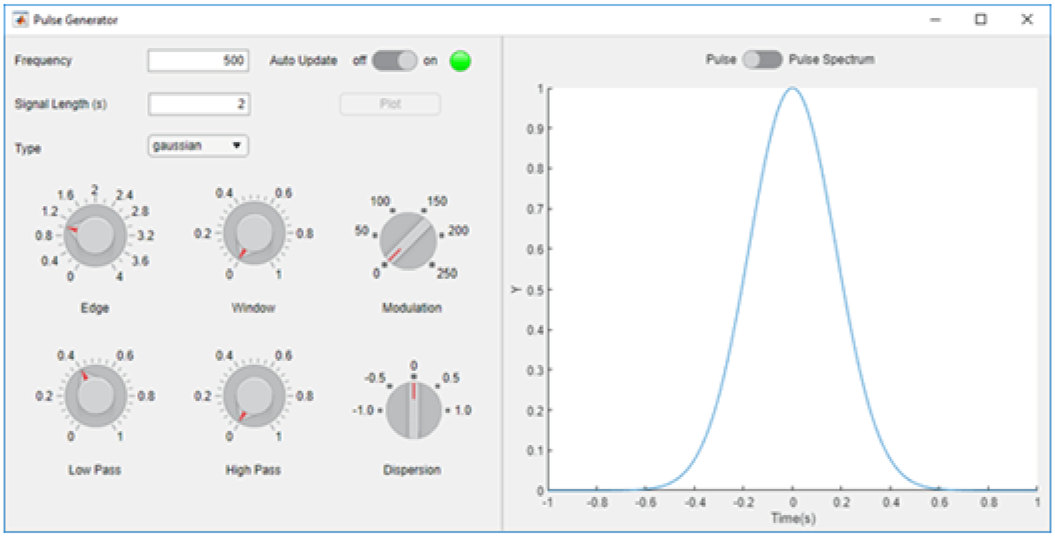
```
p = generatePulse(app.PulseObject);
```

Finally, ensure that the app calls the newly defined `generatePulse` function in the `Pulse` class by deleting the `generatePulse` function that is defined in App Designer.

To view the complete app code, see “Pulse Generator App That Stores Data in a Class” on page 8-6.

Pulse Generator App That Stores Data in a Class

This example shows the final `PulseGenerator` app, with the app data and algorithms implemented separately in the `Pulse` class. Run the example by clicking the **Run** button in App Designer.



Frequency 500

Signal Length (s) 2

Type gaussian

Auto Update off on

See Also

Related Examples

- “Role of Classes in MATLAB”
- “Components of a Class”

Create Responsive Apps

To create apps that respond quickly and smoothly to user input, you can use several performance improvement techniques in your code. These techniques include only loading and updating the parts of your app that are visible and taking advantage of certain app building capabilities that are optimized for responsiveness. Use any techniques that are helpful for the type of apps that you create and the user experience you want to provide.

- "Improve Startup Time" on page 8-8 — Load only the app content that is visible on startup. This technique can be useful if your app contains multiple tabs or large tree UI components.
- "Improve Update Time" on page 8-8 — Perform updates and calculations only when they are needed. This technique can be useful if your app contains `ValueChangingFcn` callbacks or code that updates the data in a table UI component.
- "Improve Resize Behavior" on page 8-9 — Use a grid layout manager to manage app resize behavior. This technique can be useful if your app uses a `SizeChangedFcn` callback or the `Position` property to resize UI components.
- "Improve Responsiveness to User Input" on page 8-10 — Execute a response to user input as soon as possible. This technique can be useful if your app waits for user input by using a `while` loop, or if your app performs expensive calculations that leave the interface unresponsive.
- "Improve Performance of Graphics in Your App" on page 8-11 — Fix performance bottlenecks caused by intensive plotting and data exploration. These techniques can be useful if your app contains animations or interactive plots and charts.

Improve Startup Time

When you start up an app, your code performs many tasks to load the app content. These tasks can include creating UI components, setting component properties, processing data, and performing setup calculations. As apps grow larger, these tasks can take more time, which results in longer app startup times. You can improve the startup time of your app by initializing and performing calculations for only the parts of your app that are visible at startup. You can then use callbacks to initialize and update other portions of the app after the app is loaded, and only when the app user needs to see them.

Some types of apps where this technique can have significant benefits include:

- Apps with multiple tabs — Initialize and update only the content in the tab that is visible.
- Apps containing trees with many nodes — Create child nodes only after an app user expands a parent node in the tree.

For more information, and for examples of how to update your app code in these cases, see "Improve App Startup Time" on page 8-12.

Improve Update Time

Apps often contain callbacks that update the app in response to user input. To improve responsiveness while your app is running, minimize the number of updates made in the app code. When your app performs updates and calculations only when they are needed, interactions and animations in the app can feel much smoother.

Use ValueChangedFcn Callbacks Instead of ValueChangingFcn Callbacks

Many components, such as sliders and text areas, have both a `ValueChangedFcn` callback and a `ValueChangingFcn` callback. Both of these callbacks execute in response to a change in the component value, but they execute at different times in the interaction.

- The `ValueChangedFcn` callback executes once after the app user finishes the interaction. For example, the `ValueChangedFcn` callback of a slider executes after the user releases the slider thumb at its final value.
- The `ValueChangingFcn` callback executes multiple times at regular intervals while the app user performs the interaction. For example, the `ValueChangingFcn` callback of a slider executes regularly as the user drags the slider thumb.

Using a `ValueChangedFcn` callback minimizes the number of times the callback function is executed, which can make the interaction with the component feel more responsive. Consider using a `ValueChangedFcn` over a `ValueChangingFcn` callback in these scenarios:

- There is no need to update the app until the user reaches a final value.
- Your callback function performs updates or calculations that take a long time to run.

Minimize Table Data Updates

Apps often use table UI components to store and display large amounts of data. As a result, updating that data can be an expensive operation. Improve the performance of your app when updating table data by minimizing the number of times you update the `Data` property of the `Table` object.

For example, to update two columns in a table UI component, use this code to modify the table `Data` property in a single operation instead of using a separate operation for each column:

```
fig = uifigure;
data = readtable('tsunamis.xlsx');
tbl = uitable(fig,"Data",data);

newcols = tbl.Data{:,1:2} + 1;
tbl.Data{:,1:2} = newcols;
```

Improve Resize Behavior

When a user resizes an app, it is common for all of the UI components in the app to resize in response to the new app window size. To improve resize performance in your app, consider using a grid layout manager instead of setting the `Position` property or writing a `SizeChangedFcn` callback. You can add a grid layout manager to your app by using the `uigridlayout` function or, in an existing app in App Designer, by right-clicking the canvas and selecting **Apply Grid Layout**.

Some benefits of using a grid layout manager are:

- The app manages the resize behavior without additional resize code.
- The resize operation is applied smoothly, with all components being resized at the same time.

For more information about using a grid layout manager, see “Manage App Resize Behavior Programmatically” on page 10-10.

Improve Responsiveness to User Input

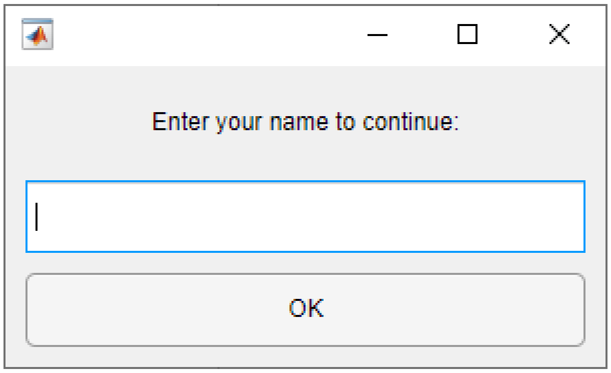
To improve the time it takes for your app to respond after a user interacts with the app interface, ensure that MATLAB executes your code that responds to the interaction as soon as possible.

Wait for User Input Using waitfor

To pause app execution while waiting for user input, use the `waitfor` function. This technique allows your app to respond to an interaction immediately and also makes the app code more readable.

For example, this code creates a dialog box that prompts a user to enter their name. Call the `waitfor` function to block code execution until the `UserData` property of the button is set to `"Clicked"`. Then update the `UserData` property in the `ButtonPushedFcn` callback. When the user enters their name and clicks the **OK** button, the code execution resumes.

```
fig = uifigure("Position",[500 500 300 150]);
gl = uigridlayout(fig,[3 1]);

lbl = uilabel(gl,"Text","Enter your name to continue:", ...
    "HorizontalAlignment","center");
ef = uieditfield(gl);
btn = uibutton(gl,"Text","OK","ButtonPushedFcn",@updateButton);

waitfor(btn,"UserData","Clicked");
disp("Program execution resumed")


function updateButton(src,event)
src.UserData = "Clicked";
end
```


Enter your name to continue:

Run App Calculations in the Background

When you run calculations in your app, the user interface can become unresponsive while MATLAB is busy. For example, MATLAB will not process callbacks in response to user interaction while a calculation is in progress. To enable your app to immediately respond to interactions even while running calculations, use the background pool to run the calculations in the background.

For an example of how to create an app that responds to button pushes while running calculations in the background, see “Create Responsive Apps by Running Calculations in the Background”. For an example of how to update a wait bar while app calculations are running, see “Update Wait Bar While Functions Run in the Background”.

Improve Performance of Graphics in Your App

If your app includes graphics, there are additional techniques that you can use to optimize performance and responsiveness:

- Update only changed data.
- Identify bottlenecks in your code.
- Limit updates to long-running animations.
- Use built-in axes interactions, and disable the interactions that the app does not require.

To learn more, see “Improve Graphics Performance”.

See Also

Related Examples

- “Improve App Startup Time” on page 8-12
- “Techniques to Improve Performance”
- “Improve Graphics Performance”
- “Profile Your Code to Improve Performance”

Improve App Startup Time

There are multiple techniques that you can use to improve the performance and responsiveness of apps that you create. For an overview of these techniques, see “Create Responsive Apps” on page 8-8.

To improve the startup time of your app, one technique is to initialize and perform calculations for only the parts of your app that are visible at startup. Two common scenarios in which this technique can have a significant benefit are when your app contains multiple tabs and when your app contains a tree with many nodes.

Improve Startup Time in Apps with Multiple Tabs

In apps with multiple tabs, only the content in a single tab is visible when the app first starts up. MATLAB optimizes startup time in apps with multiple tabs by prioritizing creating the content in the visible tab when the app first runs.

If you have a large app with many UI components, you can improve the startup time of your app by limiting the number of components that are in the visible tab. For example, consider creating a simple summary tab for your app that is visible when the app starts up. When a user runs the app, MATLAB prioritizes displaying the content in the summary tab, which allows the user to view and interact with the app sooner. You can further optimize your app layout by limiting the number of components in *each* tab, and instead using a larger number of tabs to group related app elements. This improves app responsiveness when a user switches between tabs.

Before R2022b, instead reduce startup time by populating content in tabs as the app user switches to them. For details, see Improve App Startup Time (R2022a).

Improve Startup Time in Apps with Large Trees

When you create a tree UI component with many nodes, you can provide a more responsive experience for the app user by creating child nodes only after a parent node is expanded. Do this by writing a `NodeExpandedFcn` callback for the tree, and create the nodes in the callback function.

For example, create an app that displays patient names and hospitals in a tree. Create a file named `patientTreeApp.m` in your current folder and define a function named `patientTreeApp`. Within the function, perform these steps:

1. Read in the sample patient data and store it in a table variable named `T`.
2. Create a figure window, and then create a tree in the figure.
3. Populate the tree with two top-level nodes. These nodes will have child nodes with hospital names and patient names from the data. Use the `NodeData` property to store whether the node has been expanded by a user.
4. For each of the top-level nodes, create one child node with the text `"Loading..."`. This placeholder child node allows the top-level node to be expanded by the app user. It also provides the app user with immediate feedback when they first expand a node.
5. Assign the `createNodes` function to the `NodeExpandedFcn` callback property of the tree. Pass the patient table data as an input to the function. MATLAB executes the `createNodes` function whenever the app user expands a node of the tree.

```
function patientTreeApp
T = readtable("patients");
```

```
fig = uifigure;
tr = uitree(fig,"Position",[20 20 300 300]);
hospitalnode = uitreenode(tr,"Text","Hospitals","NodeData",false);
namenode = uitreenode(tr,"Text","Patient Names","NodeData",false);
for k = 1:length(tr.Children)
    node = tr.Children(k);
    uitreenode(node,"Text","Loading...");
end

tr.NodeExpandedFcn = {@createNodes,T};
end
```

In the same file, define the `createNodes` callback function. The function input arguments are the callback source component and event data that MATLAB passes to callback functions and the patient data. Within the function, store the node that was expanded in a variable named `parent`. If this is the first time a user has expanded the node, perform these steps:

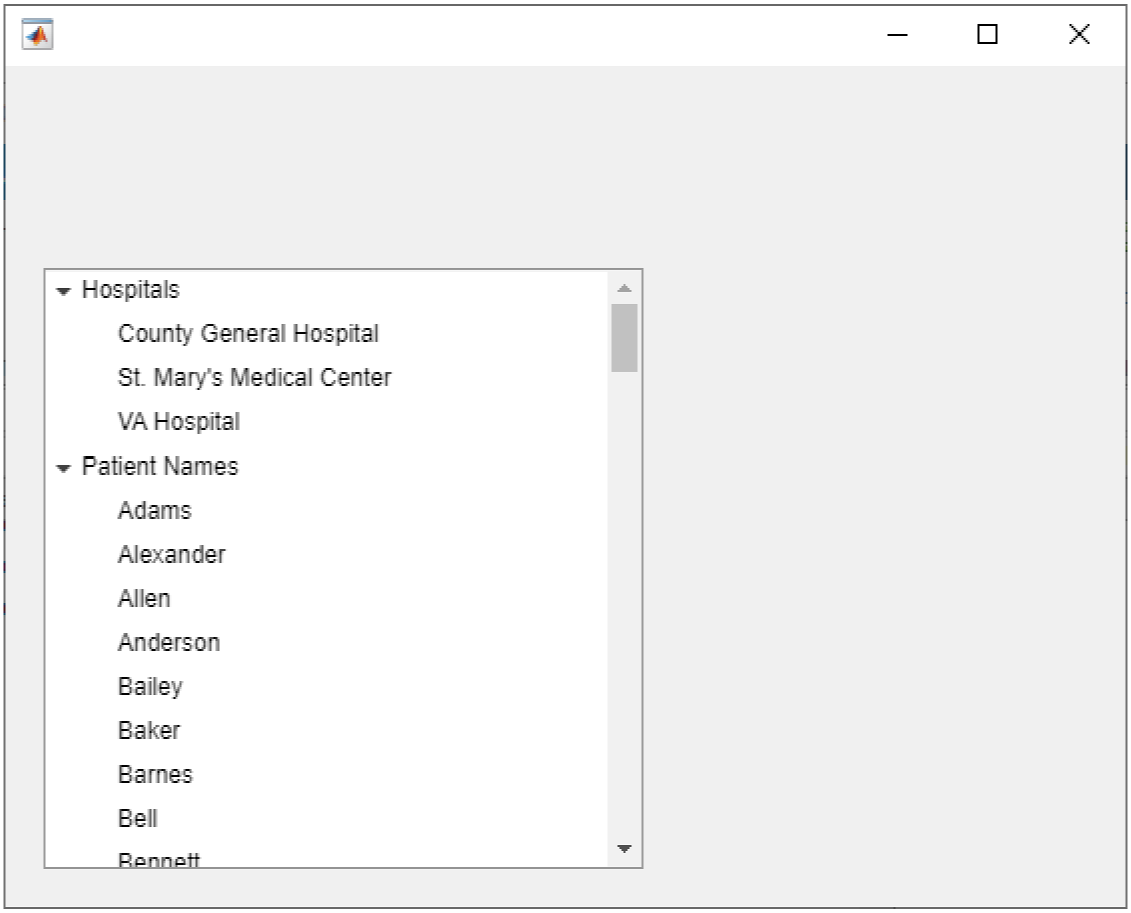
1. Delete the placeholder child node.
2. Depending on which node was expanded, store either the hospital names or the patient names in a variable named `children`.
3. For each of the names stored in `children`, create a tree node whose text is that name, and then parent it to the expanded node.
4. Update the `NodeData` property to indicate that the node has been expanded.

```
function createNodes(src,event,T)
parent = event.Node;
if ~parent.NodeData
    delete(parent.Children)

    switch parent.Text
        case "Hospitals"
            children = categories(categorical(T.Location));
        case "Patient Names"
            children = categories(categorical(T.LastName));
    end

    for k = 1:length(children)
      text = children{k};
      uitreenode(parent,"Text",text);
    end

    parent.NodeData = true;
end
end
```

Call the `patientTreeApp` function from the command line to run the app.

```
patientTreeApp
```

Expand the `Hospitals` and `Patient Names` nodes to generate and display their children.



- Hospitals
  - County General Hospital
  - St. Mary's Medical Center
  - VA Hospital
- Patient Names
  - Adams
  - Alexander
  - Allen
  - Anderson
  - Bailey
  - Baker
  - Barnes
  - Bell
  - Bennett

See Also

Related Examples

- “Create Responsive Apps” on page 8-8
- “Improve Graphics Performance”
- “Techniques to Improve Performance”

Find and Create UI Components and Charts

MATLAB provides a large set of UI components on page 4-2 and plot types for you to use when creating apps. To expand this set, users can create and share their own custom UI components and charts. If you are looking for additional UI components and charts, you can explore and download community-authored content on File Exchange at MATLAB Central. In addition, you can create your own specialized UI components and charts and share them with others.

Find Community-Authored Components and Charts

In addition to the existing UI components and charts and the examples provided in the documentation, you can find a variety of community-authored content on File Exchange at MATLAB Central. Select an entry to view additional information about the content, such as what files it includes and what documentation is available. To use a community-authored UI component or chart in your app, download the content and add it to a folder on your MATLAB search path.

Link	Options for Use in Apps
Community-authored custom UI components	• Add the UI component to your app interactively from the App Designer **Component Library**. • Add the UI component to your app programmatically by creating it in your app code.
Community-authored custom charts	• Add the chart to your app programmatically by creating it in your app code.

Create Your Own Components and Charts

MATLAB provides the ability to extend the list of available components by creating custom UI components and charts, and by embedding third-party content in your apps.

Create Custom UI Components

Create your own UI components to use in your apps or to share with others. You can use custom UI components to extend existing UI component functionality, to break up large apps into independent and maintainable pieces, and to design an interface for users to customize and reuse a component across multiple apps. For more information, see “Create a Simple Custom UI Component in App Designer” on page 13-2.

Create Custom Chart Classes

Develop your own chart class to extend existing chart functionality and to reuse and share a custom chart across multiple apps. Define a chart class by creating a subclass of the `ChartContainer` base class, and then programmatically create an instance of the chart in your app code. For more information, see “Chart Development Overview”.

Interface with Third-Party Libraries

Create an HTML UI component to embed HTML, JavaScript, or CSS content in your app. You can use the component to interface with third-party libraries to display content like widgets or data visualizations. For more information, see “Create HTML Content in Apps” on page 4-23.

See Also

Classes
`matlab.ui.componentcontainer.ComponentContainer` |
`matlab.graphics.chartcontainer.ChartContainer`

Functions
`uihtml`

Related Examples

- “Configure Custom UI Components for App Designer” on page 12-17
- “Display HTML Elements Styled by a Cascading Style Sheet” on page 7-15

9

Keyboard Shortcuts

App Designer Keyboard Shortcuts

In this section...

Shortcuts Available Throughout App Designer

Action	Keys
Run the active app.	**F5**
Save the active app.	**Ctrl+S**
Save the active app, allowing you to specify a new file name. (Save as)	**Ctrl+Shift+S**
Open a previously saved app.	**Ctrl+O**
Open a new blank app.	**Ctrl+N**
Redo an undone modification, returning it to the changed state.	**Ctrl+Y** or, in the design area only, **Ctrl+Shift+Z**
Undo a modification, returning it to the previous state.	**Ctrl+Z**
Alternate between design and code view.	**Shift+F7** If debugging is in progress, this shortcut does not change the view.
Close the active app.	**Ctrl+W**
Quit App Designer.	**Ctrl+Q**

Component Browser Shortcuts

These shortcuts are available in the **Component Browser**, in both code view and design view

Action	Keys
Select multiple components.	Hold down the **Ctrl** key as you click each component that you want to include in the multiselection.
Deselect a component from multiselection.	Hold down the **Ctrl** key as you click each component that you want to remove from a multiselection.
Navigate from clicked component to the previous or next component listed in the code browser.	**Up Arrow** and **Down Arrow**

Action	Keys
Edit code name of clicked component in the code browser.	**F2** on Windows® and Linux® **Enter** on Mac

Design View Shortcuts

These shortcuts are available from the App Designer design view only.

- “Add Component Shortcuts” on page 9-3
- “Component, Group, and Text Selection Shortcuts” on page 9-3
- “Group and Ungroup Components Shortcuts” on page 9-4
- “Component and Group Move Shortcuts” on page 9-4
- “Component Resize Shortcuts” on page 9-4
- “Component Copy, Duplicate, and Delete Shortcuts” on page 9-5
- “Design Area Grid Shortcuts” on page 9-5
- “Component Alignment Shortcuts” on page 9-5
- “Change Font Characteristics Shortcuts” on page 9-6
- “Menu Component Shortcuts” on page 9-6
- “Tab Component Shortcuts” on page 9-6
- “Navigate Canvas Shortcuts” on page 9-7

Add Component Shortcuts

Action	Shortcut
Add component and associated label (if any) to canvas.	Click the component and hold down the mouse key to drag the component from the **Component Library** on the left into the design area.
Add component only to canvas.	Hold down the **Ctrl** key, click the component, and drag it from the **Component Library** on the left into the design area.
Add label to component.	**Ctrl+L**

Component, Group, and Text Selection Shortcuts

Action	Keys
Move the selection to the next component, or container in the design area tab key navigation sequence.	**Tab**
Move the selection to the previous component or container in the design area tab key navigation sequence.	**Shift+Tab**

Action	Keys
Selects all components on the canvas, with one exception. If any of the components are grouped, the group is selected, not the individual components within the grouping.	**Ctrl+A**
Clear a component selection. Press again to reselect the component.	**Shift+Click** or **Ctrl+Click**
In the property editor or in-place editing, select all text in a text input field.	**Ctrl+A**
Select group containing a component.	**Alt+Click** a component

Group and Ungroup Components Shortcuts

Select the components that you want to group, and then press **Ctrl+G**. All components to be grouped must have the same parent component.

Action	Keys
Group selected components.	**Ctrl+G**
Ungroup components in selected group.	**Ctrl+Shift+G**

Component and Group Move Shortcuts

This table summarizes the keyboard shortcuts for moving selected components and groups.

Action	Keys
Move down 1 pixel.	**Down Arrow**
Move left 1 pixel.	**Left Arrow**
Move right 1 pixel.	**Right Arrow**
Move up 1 pixel.	**Up Arrow**
Move down 10 pixels.	**Shift+Down Arrow**
Move left 10 pixels.	**Shift+Left Arrow**
Move right 10 pixels.	**Shift+Right Arrow**
Move up 10 pixels.	**Shift+Up Arrow**
Cancel an in-progress operation.	**Escape**

Component Resize Shortcuts

Action	Keys
Resize component while maintaining aspect ratio.	Press and hold down the **Shift** key before you begin to drag the component resize handle.
Resize component while keeping center location unchanged.	Press and hold down the **Ctrl** key before you begin to drag the component resize handle.
Resize component while maintaining aspect ratio and keeping center location unchanged.	Press and hold down the **Ctrl** and **Shift** keys before you begin to drag the component resize handle.

Action	Keys
Cancel an in-progress resize operation.	**Escape**

Component Copy, Duplicate, and Delete Shortcuts

Action	Keys
Copy selected components and groups to the clipboard.	**Ctrl+C**
Duplicate the selected components and groups (without copying them to the clipboard).	**Ctrl+D**, or hold down the **Ctrl** key and drag the component.
Cut the selected components and groups from the design area onto the clipboard.	**Ctrl+X**
Delete the selected components and groups from the design area.	**Backspace** or **Delete**
Paste components and groups from the clipboard into the design area or a container component (panel, tab, or button group). Radio buttons and toggle buttons can only be pasted into radio button groups or toggle button groups, respectively.	**Ctrl+V**

Design Area Grid Shortcuts

Action	Keys
Toggle grid on and off.	**Alt+G**
Toggle snap to grid on and off.	**Alt+P**
Increase grid interval by 5 pixels.	**Alt+Page Up**
Decrease grid interval by 5 pixels.	**Alt+Page Down**

Component Alignment Shortcuts

Action	Keys
Align selected components and groups on their left edges.	**Ctrl+Alt+1**
Align selected components and groups on their horizontal centers.	**Ctrl+Alt+2**
Align selected components and groups on their right edges.	**Ctrl+Alt+3**
Align selected components and groups on their top edges.	**Ctrl+Alt+4**
Align selected components and groups on their vertical middle.	**Ctrl+Alt+5**
Align selected components and groups on their bottom edges.	**Ctrl+Alt+6**

Change Font Characteristics Shortcuts

Action	Keys
Toggle the font weight of selected components *and their children* between normal and bold.	**Ctrl+B**
Toggle the font angle of selected components *and their children* between normal and italic.	**Ctrl+I**
Decrease the value of the `FontSize` property of the selected components *and their children* by one step. Font size steps are: 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36, 48, 72.	**Ctrl+[**
Increase the value of the `FontSize` property of the selected components *and their children* by one step. Font size steps are: 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36, 48, 72.	**Ctrl+]**

Menu Component Shortcuts

Action	Keys
Add a menu item below the current item. The new menu item appears at the end of the list.	**Enter**
Add an item to the right of selected item.	**Shift+Enter**
Delete the current item.	**Delete**
Commit text changes and navigate to the next item.	Any **Arrow** key
Select the first or last item at the level of the selected item.	**Home** **End**
Move the selected child menu item higher or lower in the list.	**Ctrl+Shift+Up Arrow** **Ctrl+Shift+Down Arrow**
Move the selected top-level menu item to the left or right.	**Ctrl+Shift+Left Arrow** **Ctrl+Shift+Right Arrow**
Move the selected item to the beginning or end of the list.	**Ctrl+Shift+Home** **Ctrl+Shift+End**

Tab Component Shortcuts

Action	Keys
Move the selected tab to the left or right.	**Ctrl+Shift+Left Arrow** **Ctrl+Shift+Right Arrow**
Move the selected tab to the beginning or end.	**Ctrl+Shift+Home** **Ctrl+Shift+End**

Navigate Canvas Shortcuts

Action	Keys
Zoom in on the canvas.	**Ctrl+Plus (+)**
Zoom out on the canvas.	**Ctrl+Minus (-)**
Reset the canvas zoom to default.	**Ctrl+Alt+0**
Zoom to fit the canvas to the view.	**Space**
Pan on the canvas.	Click and drag with the middle mouse button, or hold **Space** while clicking and dragging with the left mouse button.

Code View Shortcuts

These shortcuts are available only from the App Designer code view, within the editor.

- “Code Indenting Shortcuts” on page 9-7
- “Code Folding Shortcuts” on page 9-7
- “Cut, Copy, and Paste Code Shortcuts” on page 9-8
- “Find Code Shortcuts” on page 9-8
- “Code Browser Shortcuts” on page 9-8
- “Code View Zoom Shortcuts” on page 9-8
- “Comment Shortcuts” on page 9-8
- “Bookmark Shortcuts” on page 9-8
- “Debugging Shortcuts” on page 9-9
- “Other App Designer Code Editor Shortcuts” on page 9-9

Code Indenting Shortcuts

Action	Keys
Smart indent selected code.	**Ctrl+I**
Increase indent on current line of code or currently selected code.	**Ctrl+]**
Decrease indent on current line of code or currently selected code.	**Ctrl+[**

Code Folding Shortcuts

Action	Keys
Collapse code section containing selected code.	**Ctrl+Period (.)**
Expand code section containing selected code.	**Ctrl+Shift+Period (.)**
Collapse all code sections.	**Ctrl+Comma (,)**
Expand all code sections.	**Ctrl+Shift+Comma (,)**

Cut, Copy, and Paste Code Shortcuts

Action	Keys
Cut selected code.	**Ctrl+X**
Copy selected code.	**Ctrl+C**
Paste selected code.	**Ctrl+V**
Duplicate selected lines.	**Ctrl+Shift+C**

Find Code Shortcuts

Action	Keys
Find.	**Ctrl+F**
Find next.	**F3**
Find previous.	**Shift+F3**
Find selection.	**Ctrl+F3**

Code Browser Shortcuts

Action	Keys
Delete callback.	**Delete**
Rename callback.	**F2**
Bring callback to focus and insert cursor.	**Ctrl+D**

Code View Zoom Shortcuts

Action	Keys
Zoom in on code editor.	**Ctrl+Plus (+)**
Zoom out on code editor.	**Ctrl+Minus (-)**
Reset code editor zoom to default.	**Ctrl+Alt+0**

Comment Shortcuts

Action	Keys
Add comment to selected code.	**Ctrl+R**
Remove comment from selected code.	**Ctrl+T**
Wrap selected comments.	**Ctrl+J**

Bookmark Shortcuts

Action	Keys
Set or clear bookmark.	**Ctrl+F2**
Navigate to next bookmark.	**F2**
Navigate to previous bookmark.	**Shift+F2**

Debugging Shortcuts

Action	Keys
Set or clear breakpoint.	**F12**
Continue running to next breakpoint.	**F5**
Run next line ("step").	**F10**
Run next line and step into function ("step in").	**F11**
Run until current function returns ("step out").	**Shift+F11**
Stop execution.	**Shift+F5**

Other App Designer Code Editor Shortcuts

Action	Keys
Convert selected code to uppercase or lowercase.	**Ctrl+Shift+A**
Print code.	**Ctrl+P**
Insert section break.	**Ctrl+Alt+Enter**
Evaluate selection.	**F9**
Open selection.	**Ctrl+D**
Go to specified line number.	**Ctrl+G**

Create UIs Programmatically

- “Lay Out Apps Programmatically” on page 10-2
- “Create Callbacks for Apps Created Programmatically” on page 11-2
- “Create and Run a Simple Programmatic App” on page 15-2
- “Callbacks for Specific Components” on page 19-14
- “Share Data Among Callbacks” on page 11-9

10

Lay Out a Programmatic UI

Lay Out Apps Programmatically

An app consists of a figure and one or more UI components that you place inside the figure. MATLAB app building tools provide many options for managing the layout of an app programmatically. For example, you can write code to specify the size and location of the figure and its components, align components with respect to one another, and specify the front-to-back component order.

Manage Figure Size and Location

A figure serves as the top-level container for every app. Use the uifigure function to create a figure configured for app building.

Update the size and the location of the figure on the app user's display by setting the Position property of the figure. Specify Position as a four-element vector in this form:

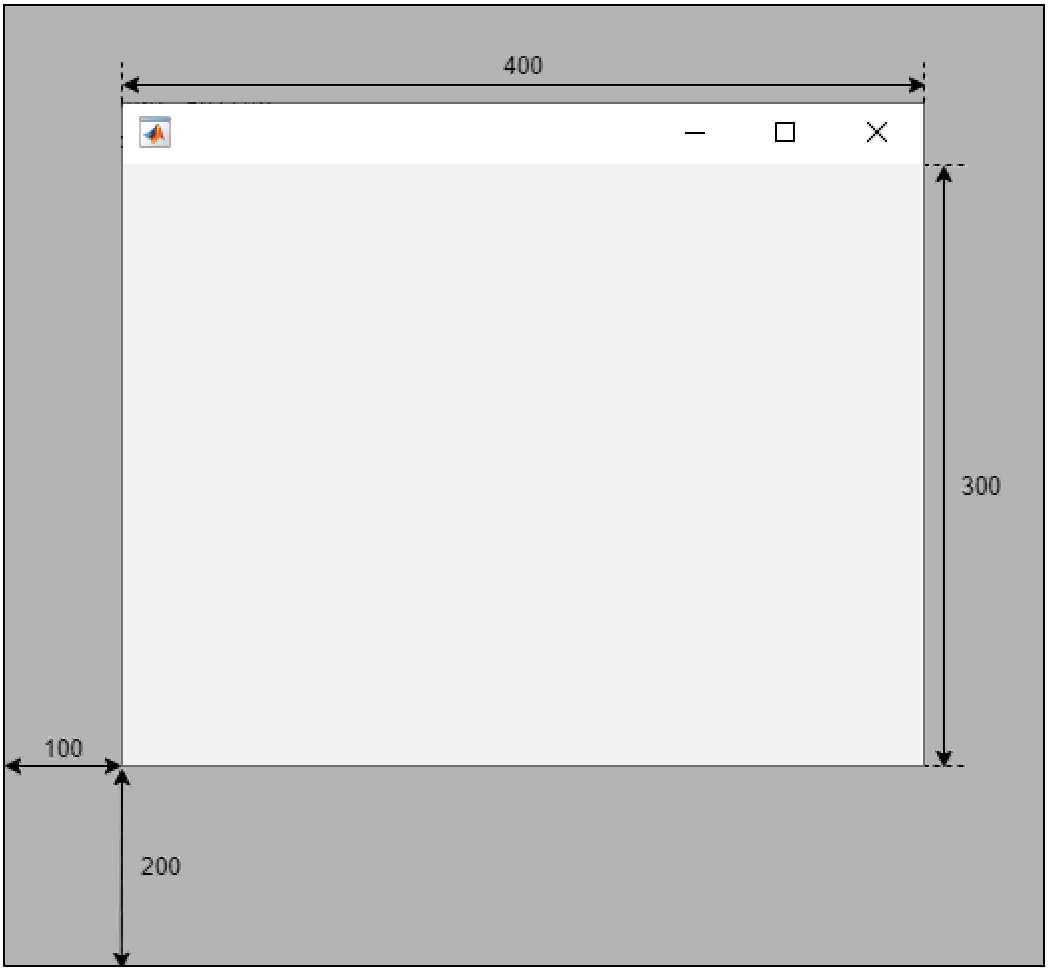
```
[left bottom width height]
```

Each element in the vector represents a distance, in pixels, that excludes the figure borders and title bar. This table describes each element.

Element	Description
left	Distance from the left edge of the primary display to the inner left edge of the figure window
bottom	Distance from the bottom edge of the primary display to the inner bottom edge of the figure window
width	Distance between the right inner and left inner edges of the figure
height	Distance between the top inner and bottom inner edges of the figure

For example, this code creates a figure window that is 100 pixels from the bottom edge and 200 pixels from the left edge of the primary display, and that is 400 pixels wide and 300 pixels tall, excluding the figure borders and title bar.

```
fig = uifigure;
fig.Position = [100 200 400 300];
```



To position a figure window in a specific location on an app user's screen, independent of the user's display size, use the `movegui` function. Specify the figure and the display location. For example, this code moves the figure window to the center of the app user's primary display.

```
movegui(fig,'center');
```

Lay Out UI Components

To design the visual appearance of your app, set the size and location of the UI components within the figure window. Lay out the components using one of these methods:

- "Use a Grid Layout Manager" on page 10-3 — Align your UI components with respect to one another, and allow the app to manage how your components resize. This method is recommended for most app building purposes.
- "Specify the Position Property" on page 10-6 — Manually position your components in the initial app layout. This method is useful when you want to specify custom resize behavior outside of the options of a grid layout manager.

Use a Grid Layout Manager

A grid layout manager is a container that lets you lay out UI components in rows and columns. Create a grid layout manager for your app using the `uigridlayout` function, and parent the grid layout manager to the main figure window. You can manage the size and configuration of the grid by setting properties of the `GridLayout` object. Add components to the grid by parenting them to the grid layout manager, and specify the row and column of each component by setting its `Layout` property.

For example, use a grid layout manager to lay out an app that contains a button, a spinner, and a text area. Give the button a fixed size, but let the other components stretch to fill the extra horizontal space. Also, center the components vertically by adding empty rows above and below them that can expand to fill the extra vertical space.

To accomplish this, create a grid with four rows and two columns by passing `[4 2]` as the second input to `uigridlayout`.

```
fig = uifigure;
fig.Position(3:4) = [300 300];
gl = uigridlayout(fig,[4 2]);
```

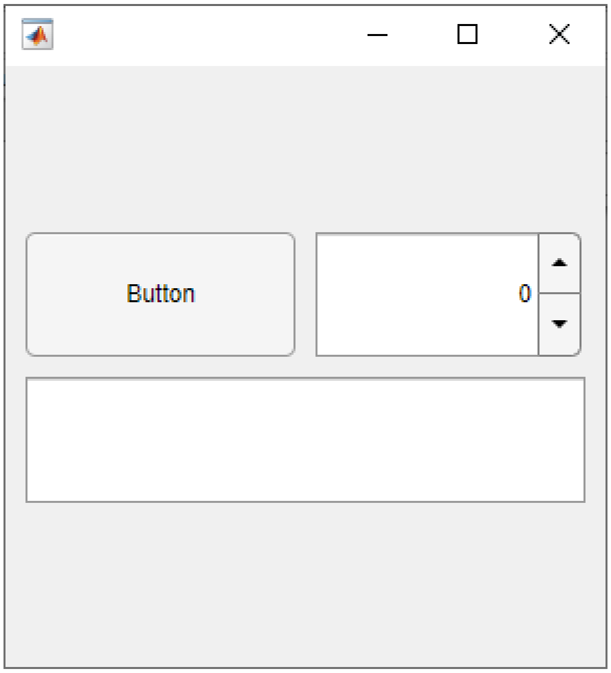
Then, create the UI components and parent them to the grid layout manager. Lay out the components using the `Layout.Row` and `Layout.Column` properties.

Position the button and the spinner next to each other by adding them to the second row.

```
btn = uibutton(gl);
btn.Layout.Row = 2;
btn.Layout.Column = 1;

spn = uispinner(gl);
spn.Layout.Row = 2;
spn.Layout.Column = 2;
```

Position the text area underneath by adding it to the third row. Lay out the text area to span both the first and second column of the grid by setting its `Layout.Column` property to `[1 2]`.

```
ta = uitextarea(gl);
ta.Layout.Row = 3;
ta.Layout.Column = [1 2];
```



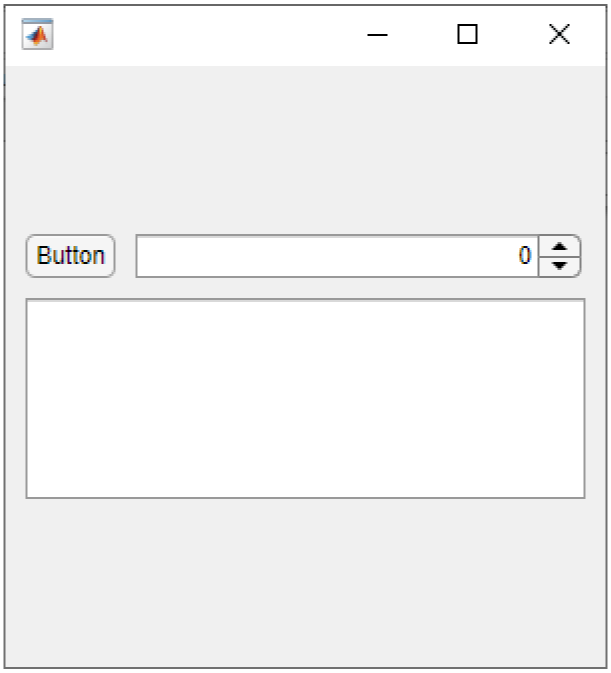
When you create a grid layout manager, by default, each row has the same height and each column has the same width. Resize and reposition the UI components by setting the `RowHeight` and `ColumnWidth` properties of the grid layout manager.

Set the height of the second row to automatically scale to fit its contents, and the height of the third row to be fixed at 100 pixels. Set the heights of the first and fourth rows to `'1x'`. This specifies that the top and bottom rows have the same height and expand to fill the remaining vertical space, which ensures the components are centered in the figure window.

```
gl.RowHeight = {'1x','fit',100,'1x'};
```

Set the width of the first column to automatically scale to fit its contents. This resizes the width of the button to fit the length of its text. Set the width of the second column to `'1x'` to fill the remaining horizontal space.

```
gl.ColumnWidth = {'fit','1x'};
```



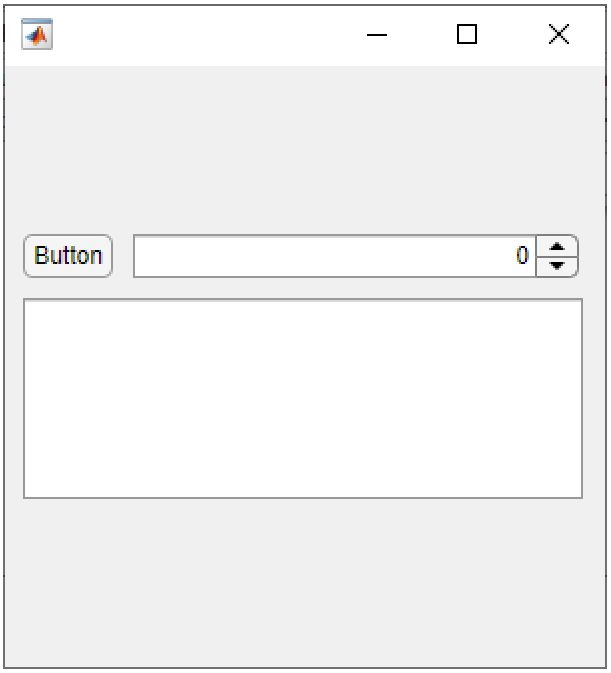
An additional benefit of using a grid layout manager is that you can use the `ColumnWidth` and `RowHeight` properties to manage how the UI components in your app resize when the app user resizes the figure window. For more information, see "Manage App Resize Behavior Programmatically" on page 10-10.

Specify the Position Property

Alternatively, you can manually position the UI components in you app. Every UI component has a `Position` property. Use this property to control the size and location of the component in the figure window. Specify the value of `Position` as a four-element vector of the form `[left bottom width height]`.

For example, use the `Position` property to lay out an app that contains a button, a spinner, and a text area. Align the button and the spinner horizontally by specifying that they have the same distance from the bottom edge of the figure and the same height. Position the text area below the button and slider, and set its width to span the width the two components above.

```
fig = uifigure;
fig.Position(3:4) = [300 300];

btn = uibutton(fig);
btn.Position = [10 195 45 22];

spn = uispinner(fig);
spn.Position = [65 195 225 22];

ta = uitextarea(fig);
ta.Position = [10 85 280 100];
```



The position of a UI component is calculated relative to the immediate parent of the component. For instance, if you create a label inside a panel, the values of `left` and `bottom` in the position vector of the `Label` object indicate the distance from the left and bottom edges of the panel, not the figure window.

Change Front-to-Back Component Order

The stacking order of UI components determines which components appear in front of other overlapping components in an app. The default stacking order of components is as follows:

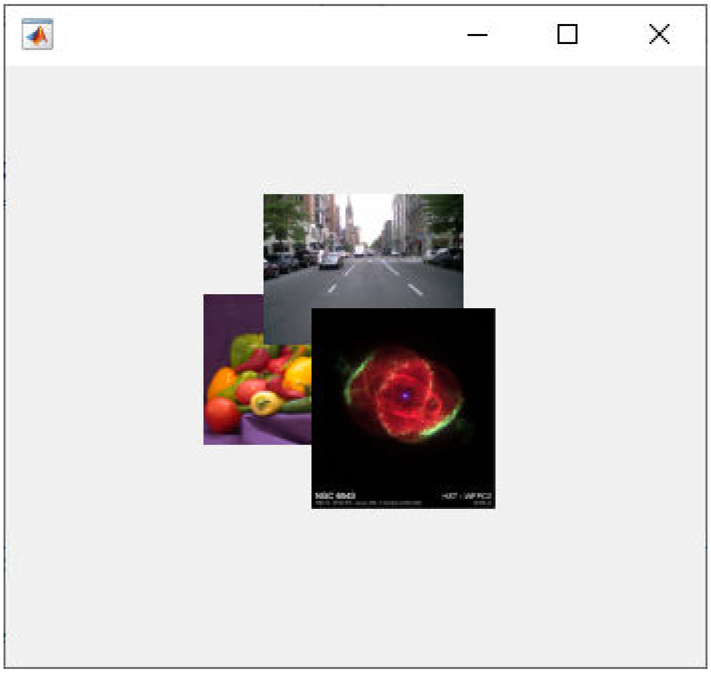
- UI components and containers appear in the order in which you create them. New components appear in front of existing components.
- Axes and other graphics objects appear behind UI components and containers.

An exception to this default order is for tabs within tab groups. The first tab created in a tab group appears on top of the other tabs.

For example, this code creates three overlapping images in a figure. The image created first is on the bottom, and the image created last is on top.

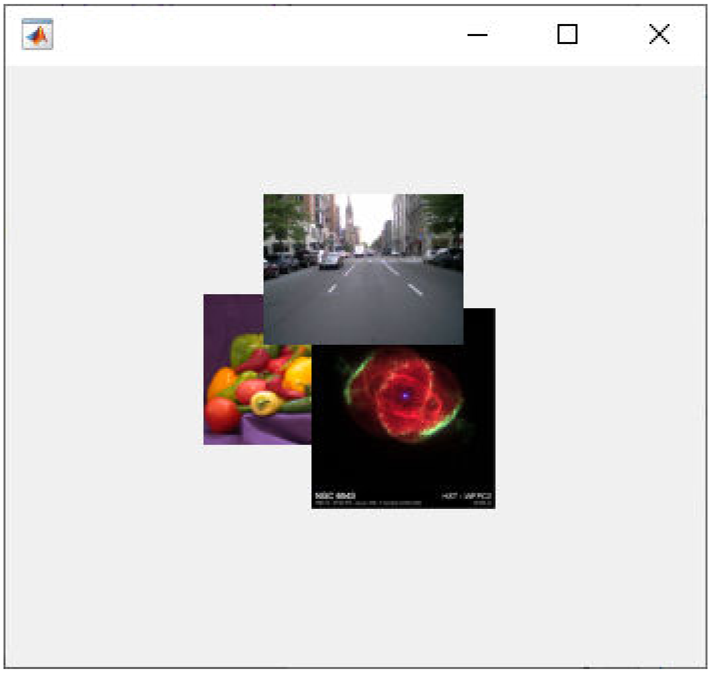
```
fig = uifigure;
fig.Position = [100 100 350 300];

peppers = uiimage(fig);
peppers.ImageSource = "peppers.png";

street = uiimage(fig);
street.ImageSource = "street1.jpg";
street.Position(1:2) = [130 150];
```

```
nebula = uiimage(fig);
nebula.ImageSource = "ngc6543a.jpg";
nebula.Position(1:2) = [150 80];
```



To modify the stacking order in your app, use the `uistack` function. For example, bring the image of the street to the top by rearranging the stacking order of the images.

```
uistack(street,"top")
```



There are some restrictions on stacking order. Axes and other graphics objects can stack in any order with respect to one another. However, axes and other graphics objects always appear behind UI components and containers.

You can work around this restriction by parenting graphics objects to separate containers. Then, you can stack those containers in any order. To parent a graphics object to a container, set its `Parent` property to be that container. For example, you can parent an `Axes` object to a panel by setting the `Parent` property of the `Axes` to be the panel.

See Also

Related Examples

- "Create Callbacks for Apps Created Programmatically" on page 11-2
- "Manage App Resize Behavior Programmatically" on page 10-10
- "Lay Out Apps in App Designer Design View" on page 5-2

Manage App Resize Behavior Programmatically

Apps created using the uifigure function are resizable by default. The components reposition and resize automatically as the app user changes the size of the window at run-time.

If you want more flexibility over how your app resizes, use one of these methods:

- “Use a Grid Layout Manager” on page 10-10 — Add components to a grid, and specify how the rows and columns of the grid resize.
- “Write Code to Manage Resize Behavior” on page 10-12 — Write a SizeChangedFcn callback that resizes UI components. The callback executes whenever the figure window size changes.
- “Turn Off Resizing of Specific Components” on page 10-15 — Specify the AutoResizeChildren property of specific containers in your app.
- “Turn Off App Resizing Entirely” on page 10-16 — Set the Resize property of the figure to 'off'.

Use a Grid Layout Manager

A grid layout manager is a container that allows you to lay out UI components in a grid. You can configure grid layout managers to specify the initial layout and resize behavior of the components in the grid.

Create a grid layout manager in a UI figure window by calling the uigridlayout function and specifying the figure as the first argument. Set the RowHeight and ColumnWidth properties of the grid layout manager to specify how each row and column behaves when the app user resizes the figure window. Specify RowHeight and ColumnWidth as a cell array with one value for each row or column. There are three different types of row heights and column widths:

- Fit size — Specify 'fit'. The row height or column width is fixed to automatically fit its contents. The dimension does not change when the app is resized.
- Fixed size — Specify a number in pixels. The row height or column width is fixed at the number of pixels you specify. The dimension does not changed when the app is resized.
- Variable size — Specify a number paired with an 'x' character (for example, '1x'). Variable-height rows fill the remaining vertical space that the fixed-height rows do not use, and variable-width columns fill the remaining horizontal space that fixed-width rows do not use. The number you pair with the 'x' character is a weight for dividing up the remaining space, where the amount of space is proportional to the number. For instance, if one row has a width of '2x' and another has a width of '1x', the first row grows or shrinks twice as much as the second when the app is resized.

For example, this code creates a grid layout manager with four rows. The height of the first row is sized to fit its content, the second row is fixed at 200 pixels, and the last two rows share the remaining vertical space unequally. The third row uses twice as much space as the fourth row.

```
fig = uifigure;
gl = uigridlayout(fig,[4 1]);
gl.RowHeight = {'fit',200,'2x','1x'};
```

For more information about laying out apps using a grid layout manager, see “Lay Out Apps Programmatically” on page 10-2.

Example: Resizable App Using a Grid Layout Manager

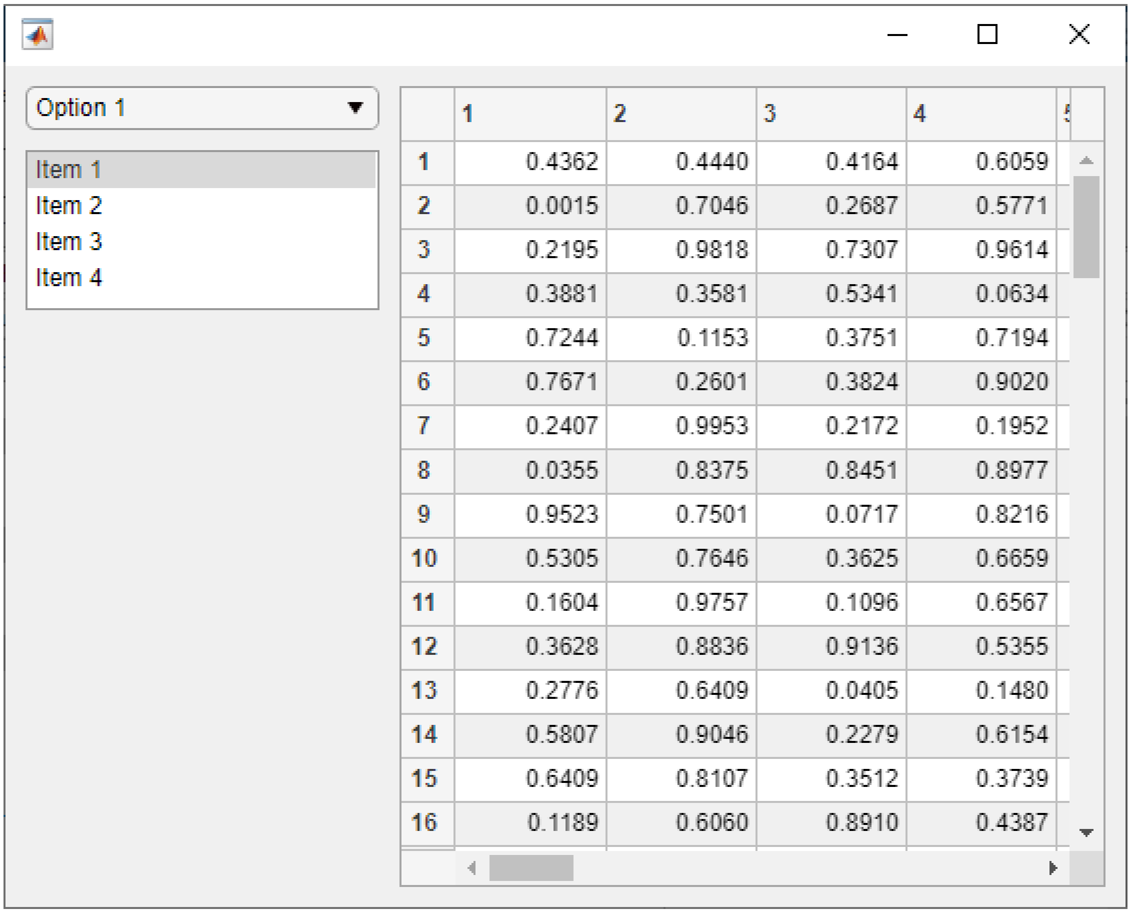
This example demonstrates how to configure a grid layout manager to specify app resize behavior. The app contains a drop-down, a list box, and a table with some data. Create a UI figure window with a 3-by-2 grid layout. Then, create the UI components and add them to the grid layout by specifying the `Layout.Row` and `Layout.Column` properties.

```
fig = uifigure;
gl = uigridlayout(fig,[3 2]);

dd = uidropdown(gl);
dd.Layout.Row = 1;
dd.Layout.Column = 1;

lb = uilistbox(gl);
lb.Layout.Row = 2;
lb.Layout.Column = 1;

tbl = uitable(gl);
tbl.Data = rand(100);
tbl.Layout.Row = [1 3];
tbl.Layout.Column = 2;
```

Configure the app layout and resize behavior by setting the `RowHeight` and `ColumnWidth` properties of the grid layout manager:

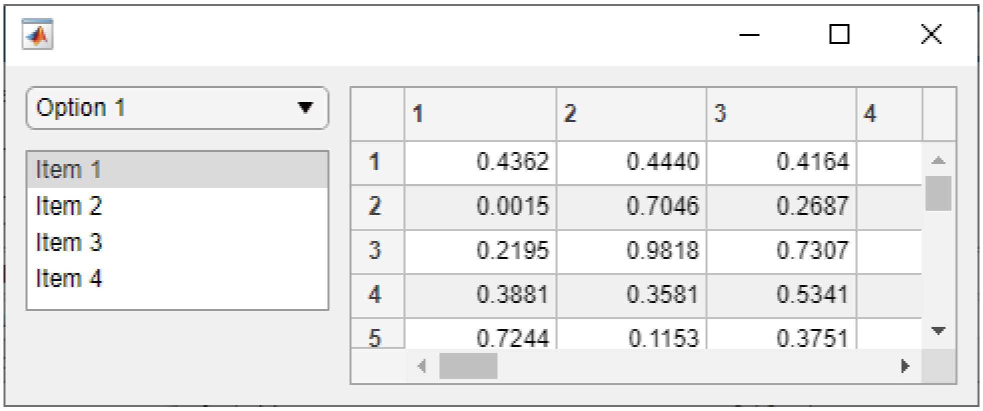
- Specify `'fit'` for the first row. This automatically adjusts the row height to fit the height of the drop-down.
- Specify a height of 80 pixels for the second row. This fixes the list box height when the app is resized.
- Specify a height of `'1x'` for the third row. This fills the remaining vertical space.
- Specify a width of `'1x'` for the first column and `'2x'` for the second. This ensures that all components resize horizontally, and the table always occupies twice as much horizontal space as the other components.

```
gl.RowHeight = {'fit',80,'1x'};
gl.ColumnWidth = {'1x','2x'};
```



	1	2	3	4
1	0.4362	0.4440	0.4164	0.6059
2	0.0015	0.7046	0.2687	0.5771
3	0.2195	0.9818	0.7307	0.9614
4	0.3881	0.3581	0.5341	0.0634
5	0.7244	0.1153	0.3751	0.7194
6	0.7671	0.2601	0.3824	0.9020
7	0.2407	0.9953	0.2172	0.1952
8	0.0355	0.8375	0.8451	0.8977
9	0.9523	0.7501	0.0717	0.8216
10	0.5305	0.7646	0.3625	0.6659
11	0.1604	0.9757	0.1096	0.6567
12	0.3628	0.8836	0.9136	0.5355
13	0.2776	0.6409	0.0405	0.1480
14	0.5807	0.9046	0.2279	0.6154
15	0.6409	0.8107	0.3512	0.3739
16	0.1189	0.6060	0.8910	0.4387

Resize the figure window by dragging one of the window corners. The UI components resize according to the grid layout specifications.



	1	2	3	4
1	0.4362	0.4440	0.4164	
2	0.0015	0.7046	0.2687	
3	0.2195	0.9818	0.7307	
4	0.3881	0.3581	0.5341	
5	0.7244	0.1153	0.3751	

Write Code to Manage Resize Behavior

When you wish to provide resize behavior that the grid layout manager does not support, consider managing your app layout using `SizeChangedFcn` callbacks. For example, use this method if you want to:

- Resize a component up to a minimum or maximum size that you define.
- Implement non-linear resize behaviors.

To specify resize behavior in this way, follow these steps:

1. Write callback functions for each container in your app to manage the layout of its children when the window size changes.
2. Set the AutoResizeChildren property of each container to 'off'.
3. Set the SizeChangedFcn property of each container to a handle to the appropriate callback function.

It is a good practice to put all the layout code for each container inside the SizeChangedFcn callback to ensure the most accurate results.

The SizeChangedFcn callback executes when one of these happens:

- The container becomes visible for the first time.
- The container is visible while its size changes.
- The container becomes visible for the first time after its size changes. This occurs when the size changes while the container is invisible, and then it becomes visible later.

Tip It is a good practice to delay the display of the container until after all the variables that the SizeChangedFcn uses are defined. This practice can prevent the SizeChangedFcn callback from returning an error. To delay the display of the container, set its Visible property to 'off'. Then, set the Visible property to 'on' after you define the variables that your SizeChangedFcn callback uses.

Example: Resizable App Using SizeChangedFcn

This example demonstrates how to create an app that uses custom resize logic to manage the size of toggle buttons within a button group, and to fix the aspect ratio of a set of axes. Create a file named sizeChangedApp.m in your current folder, and define the main sizeChangedApp function at the top of the file.

Write a helper function named createComponents to create the figure and UI components, and store the UI components in the UserData of the figure. This allows you to access your app data within the figure SizeChangedFcn callback function. For more information about sharing app data, see “Share Data Among Callbacks” on page 11-9.

Call the createComponents function in your main app function, and then make the figure window visible.

```
function sizeChangedApp
  fig = createComponents;
  fig.Visible = 'on';
end

% Create UI components
function fig = createComponents
  fig = uifigure('Visible','off', ...
    'AutoResizeChildren','off', ...
    'SizeChangedFcn',@figResize);
  btngrp = uibuttongroup(fig, ...
      'AutoResizeChildren','off', ...
      'SizeChangedFcn',@bgResize);
  btn1 = uitogglebutton(btngrp);
  btn2 = uitogglebutton(btngrp);
  ax = uiaxes(fig);

  % Store components
  fig.UserData = struct(...
    'ButtonGroup',btngrp, ...
    'Button1',btn1, ...
    'Button2',btn2, ...
    'Axes',ax);
end
```

Then, write one `SizeChangedFcn` resize function for the figure window and another one for the button group. Each function manages the resize behavior of its immediate children.

For the figure window, write a callback named `figResize` to manage the location and size of the `ButtonGroup` and `Axes` objects whenever the user resizes the window:

- Position the button group to span the entire left half of the figure.
- Position the axes to maintain a square aspect ratio and a position in the center of the right half of the figure:
  - Set the width and height of the `Axes` object to be the same, with the number of pixels given by `axdim`. The value of `axdim` is the value that fills the right half of the figure to its fullest, allowing for 10 pixels of space on each size of the axes and subject to the constraint that the axes remains square. The command `axdim = max(axdim,0)` ensures the dimensions of the axes are never negative.
  - Set the left edge of the axes, `axleft`, so that the axes is horizontally centered in the right half of the figure.
  - Set the bottom edge of the axes, `axbottom`, so that the axes is vertically centered in the figure.

```
function figResize(src,event)
  % Get UserData to access components
  data = src.UserData;

  % Get figure size
  figwidth = src.Position(3);
  figheight = src.Position(4);

  % Resize button group
  data.ButtonGroup.Position = [1 1 figwidth/2 figheight];

  % Resize axes
  axdim = min(figwidth/2,figheight) - 20;
  axdim = max(axdim,0);
  axleft = figwidth/2 + (figwidth/2-axdim)/2;
  axbottom = (figheight-axdim)/2;
  data.Axes.Position = [axleft axbottom axdim axdim];
end
```

For the button group, write a callback named `bgResize` to manage the location and size of the `ToggleButton` objects. This callback is executed whenever the `ButtonGroup` object changes size, which occurs whenever the user resizes the figure window. In this function, position the two toggle buttons relative to the size of the `ButtonGroup` object:

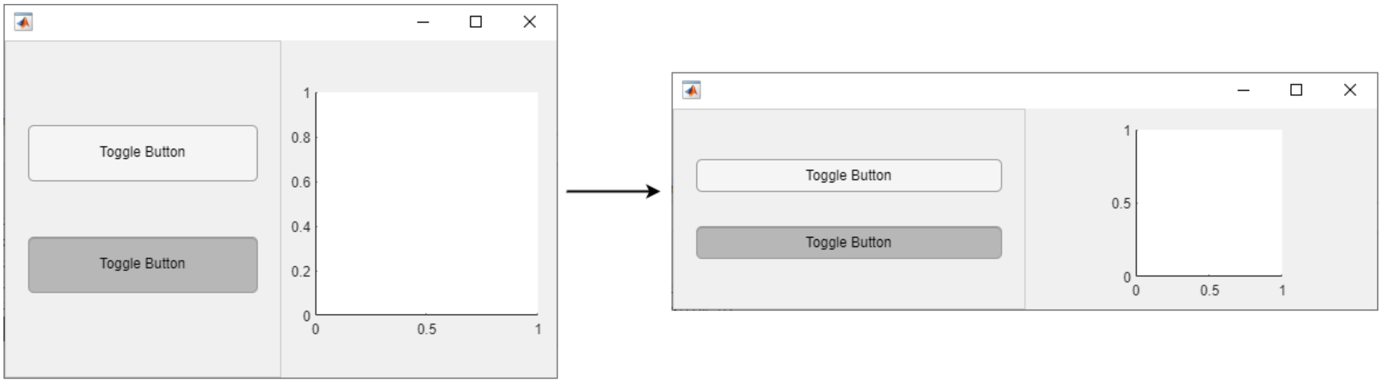
- Set the left edge and width of each toggle button, `btnleft` and `btnwidth`, to allow for 20 pixels of space between the button edges and the container edges on both the left and the right side.
- Set the height of each toggle button, `btnheight`, to 1/6 the height of the button group.
- Set the bottom of each toggle button, `btn1bottom` and `btn2bottom`, so that the space above the top button and below the bottom button is 1/4 the height of the button group.

This is the code for the app:

```
function bgResize(src,event)
  % Get UserData to access components
  fig = ancestor(src,'figure','toplevel');
  data = fig.UserData;

  % Get button group size
  bgwidth = src.Position(3);
  bgheight = src.Position(4);

  % Resize button group
  btnleft = 20;
  btn1bottom = bgheight/4;
  btn2bottom = (7/12)*bgheight;
  btnwidth = bgwidth-40;
  btnheight = bgheight/6;
  data.Button1.Position = [btnleft btn1bottom btnwidth btnheight];
  data.Button2.Position = [btnleft btn2bottom btnwidth btnheight];
end
```

Run the app, and then resize the figure window. The components in the app resize relative to the size of the figure window.

```
sizeChangedApp
```



Turn Off Resizing of Specific Components

The `AutoResizeChildren` property controls automatic resize behavior for apps without a grid layout manager or a `SizeChangedFcn` callback. Every app container, such as a UI figure, panel, or tab, has an `AutoResizeChildren` property, which is set to `'on'` by default. When a container has `AutoResizeChildren` set to `'on'`, the app automatically resizes the children of that container when the app user resizes the figure window. Use this property to selectively turn off resizing for specific components:

- To turn off automatic resizing entirely, set `AutoResizeChildren` of the main figure window to `'off'`.
- To turn off automatic resizing for specific components, parent those components to a container with `AutoResizeChildren` set to `'off'`.

When you change the `AutoResizeChildren` property of both a parent container and one of its children, first set the value for the parent container, then set it for the child container.

Turn Off App Resizing Entirely

The `Resize` property of a figure window controls whether the app user can interactively resize the window. The default value of `Resize` is `'on'`. Consider setting `Resize` to `'off'` if a consistent window size is important to the layout or behavior of your app.

See Also

`uifigure` | `uigridlayout`

Related Examples

- “Create Callbacks for Apps Created Programmatically” on page 11-2
- “Lay Out Apps Programmatically” on page 10-2
- “Manage Resizable Apps in App Designer” on page 5-12

DPI-Aware Behavior in MATLAB

In this section...

Starting in R2015b, MATLAB is DPI-aware, which means that it takes advantage of your full system resolution to draw graphical elements (fonts, UIs, and graphics). Graphical elements appear sharp and consistent in size on these high-DPI systems:

- Windows systems in which the display dots-per-inch (DPI) value is set higher than 96
- Macintosh systems with Apple Retina® displays

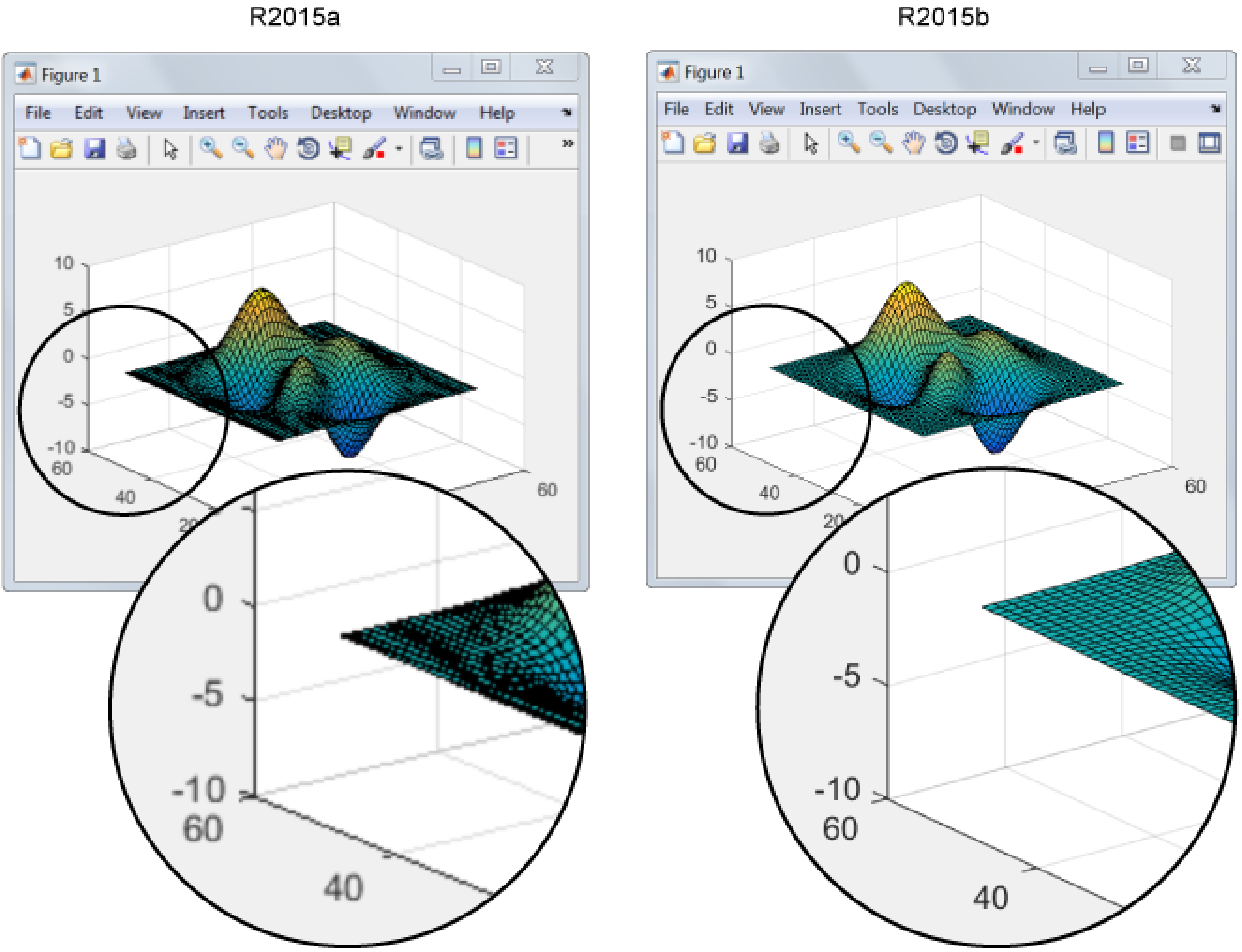
DPI-aware behavior does not apply to Linux systems.

Previously, MATLAB allowed some operating systems to scale graphical elements. That scaling helped to maintain consistent appearance and functionality, but it also introduced undesirable effects. Graphical elements often looked blurry, and the size of those elements was sometimes inconsistent.

Visual Appearance

Here are the visual effects you might notice on high-DPI systems:

- The MATLAB desktop, graphics, fonts, and most UI components look sharp and render with full graphical detail on Macintosh and Windows systems.



- When you create a graphics or UI object, and specify the `Units` as `'pixels'`, the size of that object is now consistent with the size of other objects. For example, the size of a push button (specified in pixels) is now consistent with the size of the text on that push button (specified in points).
- Elements in the MATLAB Toolstrip look sharper than in previous releases. However, icons in the Toolstrip might still look slightly blurry on some systems.
- On Windows systems, the MATLAB Toolstrip might appear larger than in previous releases.
- On Windows systems, the size of the Command Window fonts and Editor fonts might be larger than in previous releases. In particular, you might see a difference if you have nondefault font sizes selected in MATLAB preferences. You might need to adjust those font sizes to make them look smaller.
- You might see differences on multiple-display systems that include a combination of different displays (for example, some, but not all of the displays are high-DPI). Graphical elements might look different across displays on those systems.

Using Object Properties

These changes to object properties minimize the impact on your existing code and allow MATLAB to use the full display resolution when rendering graphical elements. All UIs you create in MATLAB are automatically DPI-aware applications.

Units Property

When you set the `Units` property of a graphics or UI object to `'pixels'`, the size of each pixel is now device-independent on Windows and Macintosh systems:

- On Windows systems, 1 pixel = 1/96 inch.
- On Macintosh systems, 1 pixel = 1/72 inch.
- On Linux systems, the size of a pixel is determined by the display DPI.

Your existing graphics and UI code will continue to function properly with the new pixel size. Keep in mind that specifying (or querying) the size and location of an object in pixels might not correspond to the actual pixels on your screen.

For example, each screen pixel on a 192-DPI Windows system is 1/192nd of an inch. In this case, twice as many screen pixels cover the same linear distance as the device-independent pixels do. If you create a figure, and specify its size to be 500-by-400 pixels, MATLAB reports the size to be 500-by-400 in the `Position` property. However, the display uses 1000-by-800 screen pixels to cover the same graphical region.

Note Starting in R2015b, MATLAB might report the size and location of objects as fractional values (in pixel units) more frequently than in previous releases. For example, your code might report fractional values in the `Position` property of a figure, whereas previous releases reported whole numbers for that same figure.

Root ScreenSize Property

The `ScreenSize` property of the root object might not match the display size reported by high-DPI Windows systems. Specifically, the values do not match when the `Units` property of the root object is set to `'pixels'`. MATLAB reports the value of the `ScreenSize` property based on device-independent pixels, not the size of the actual pixels on the screen.

Root ScreenPixelsPerInch Property

The `ScreenPixelsPerInch` property became a read-only property in R2015b. If you want to change the size of text and other elements on the screen, adjust your operating system settings.

Also, you cannot set or query the default value of the `ScreenPixelsPerInch` property. These commands now return an error:

```
get(groot,'DefaultRootScreenPixelsPerInch')
set(groot,'DefaultRootScreenPixelsPerInch')
```

The factory value cannot be queried either. This command returns an error as well:

```
get(groot,'FactoryRootScreenPixelsPerInch')
```

Using print, getframe, and publish Functions

getframe and print Functions

When using the `getframe` function (or the `print` function with the `-r0` option) on a high-DPI system, the size of the image data array that MATLAB returns is larger than in previous releases. Additionally, the number of elements in the array might not match the figure size in pixel units. MATLAB reports the figure size based on device-independent pixels. However, the size of the array is based on the display DPI.

publish Function

When publishing documents on a high-DPI system, the images saved to disk are larger than in previous releases or on other systems.

See Also

Root | Figure

Create and Manage Callbacks Programmatically

Create Callbacks for Apps Created Programmatically

In this section...

To program a UI component in your app to respond to an app user's input, create a callback function for that UI component. A callback function is a function that executes in response to a user interaction, such as a click on a button. Every UI component has multiple callback properties, each of which corresponds to a specific action. When a user runs your app and performs one of these actions, MATLAB executes the function assigned to the associated callback property.

For example, if your app contains a button, you might want to make the app update when a user clicks that button. You can do this by writing a function that performs the update, then setting the `ButtonPushedFcn` property of the button to a handle to your function. You can assign a callback function to a callback property as a name-value argument when you create the component, or you can set the property using dot notation from anywhere in your code.

To determine the callback properties a UI component has, see the properties page of the specific UI component.

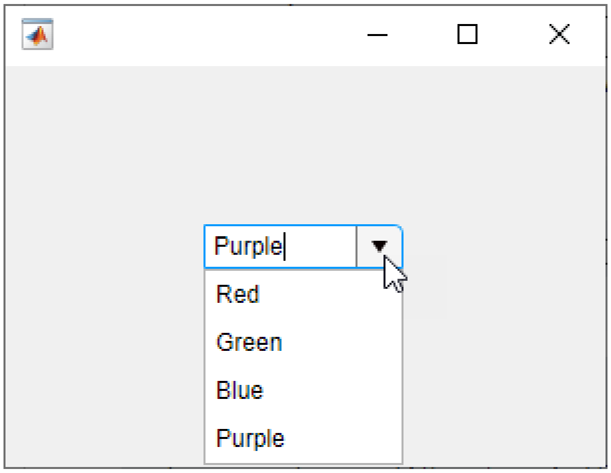
Callback Function Arguments

When a UI component executes a callback function, MATLAB automatically passes two input arguments to the function. These input arguments are often named `src` and `event`. The first argument is the UI component that triggered the callback. The second argument provides event data to the callback function. The event data that it provides is specific to the callback property and the component type. To determine the event data associated with a callback property, see the properties page of the UI component that executes the callback.

For example, the `updateDropDown` function uses these callback inputs to add items to an editable drop-down menu when the user types a new value. When the drop-down executes the `addItems` callback, `src` contains the drop-down component, and `event` contains information about the interaction. The function uses the `event.Edited` property to check if the value is a new value that the user typed, or an existing item. Then, if the value is new, the function uses the `event.Value` property to add the value to the drop-down items.

To run this function, save it to a file named `updateDropDown.m` on the MATLAB path. Type a new value in the drop-down menu, press **Enter**, and view the updated drop-down items.

```
function updateDropDown
  fig = uifigure('Position',[500 500 300 200]);
  dd = uidropdown(fig, ...
    'Editable','on', ...
    'Items',{'Red','Green','Blue'}, ...
    'ValueChangedFcn',@addItems);
end

function addItems(src,event)
  val = event.Value;
  if event.Edited
    src.Items{end+1} = val;
  end
end
```



Specify a Callback Function

Assign a callback function to a callback property in one of the following ways:

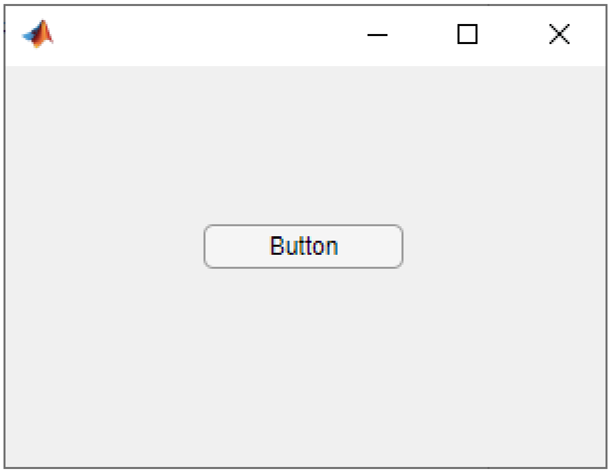
- “Specify a Function Handle” on page 11-3 — Use this method when your callback does not require additional input arguments.
- “Specify a Cell Array” on page 11-4 — Use this method when your callback requires additional input arguments. The cell array contains a function handle as the first element, followed by any input arguments you want to use in the function.
- “Specify an Anonymous Function” on page 11-5 — Use this method when your callback code is simple, or to reuse a function that is not always executed as a callback.
- “Specify Text Containing MATLAB Commands (Not Recommended)” on page 11-6

Specify a Function Handle

Function handles provide a way to represent a function as a variable. The function can be either a local or nested function in the same file as the app code, or a function defined in a separate file that is on the MATLAB path. To create the function handle, specify the `@` operator before the name of the function.

For example, to create a button that responds to a click, save the following function to a file named `codeButtonResponse.m` on the MATLAB path. This code creates a button using the `uibutton` function and sets the `ButtonPushedFcn` property to be a handle to the function `buttonCallback`. It creates this handle using the notation `@buttonCallback`. Notice that the function handle does not explicitly refer to any input arguments, but the function declaration includes the `src` and `event` input arguments.

```
function codeButtonResponse
  fig = uifigure('Position',[500 500 300 200]);
  btn = uibutton(fig,'ButtonPushedFcn',@buttonCallback);

  function buttonCallback(src,event)
    disp('Button pressed');
  end
end
```



A benefit of specifying callbacks as function handles is that MATLAB checks each callback function for syntax errors and missing dependencies when you assign it to the component. If there is a problem in a callback function, then MATLAB returns an error immediately instead of waiting for the user to trigger the callback. This behavior helps you to find problems in your code before the user encounters them.

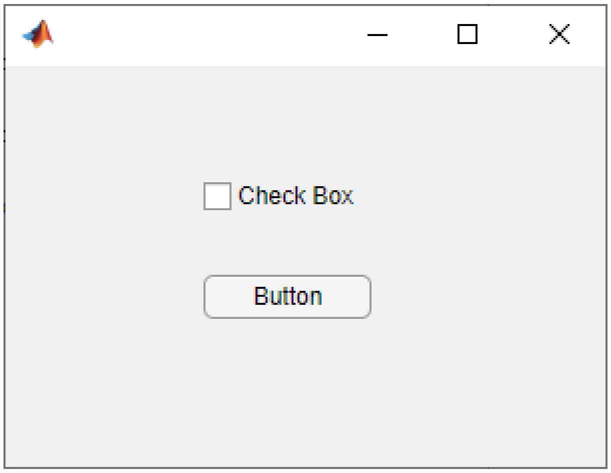
Specify a Cell Array

All callbacks accept two input arguments for the source and event. To specify a callback that accepts additional input arguments beyond these two, use a cell array. The first element in the cell array is a function handle. The other elements in the cell array are the additional input arguments you want to use, separated by commas. The function you specify must accept the source and event arguments as its first two input arguments, as described in “Specify a Function Handle” on page 11-3. However, you can define additional inputs in your function declaration after these first two arguments.

For example, the `codeComponentResponse` function creates a button and a check box component that both use the same function as a callback, but that pass different arguments to it. To specify different input arguments for the different components, set the callback properties of both components to cell arrays. The first element of the cell array is a handle to the `componentCallback` function, and the second is the additional input argument to pass to the function.

To run this example, save the function to a file named `codeComponentResponse.m` on the MATLAB path. When you select or clear the check box, MATLAB displays `You clicked the check box`. When you click the button, MATLAB displays `You clicked the button`.

```
function codeComponentResponse
  fig = uifigure('Position',[500 500 300 200]);
  cbx = uicheckbox(fig,'Position',[100 125 84 22], ...
    'ValueChangedFcn',{@componentCallback,'check box'});
  btn = uibutton(fig,'Position',[100 75 84 22], ...
    'ButtonPushedFcn',{@componentCallback,'button'});

  function componentCallback(src,event,comp)
    disp(['You clicked the ' comp]);
  end
end
```



Like callbacks specified as function handles, MATLAB checks callbacks specified as cell arrays for syntax errors and missing dependencies when you assign them to a component. If there is a problem in the callback function, then MATLAB returns an error immediately instead of waiting for the user to trigger the callback. This behavior helps you to find problems in your code before the user encounters them.

Specify an Anonymous Function

An anonymous function is a function that is not stored in a program file. Specify an anonymous function when:

- You want a UI component to execute a function that does not support the two source and event arguments that are required for function handles and cell arrays.
- You want a UI component to execute a script.
- Your callback consists of a single executable statement.

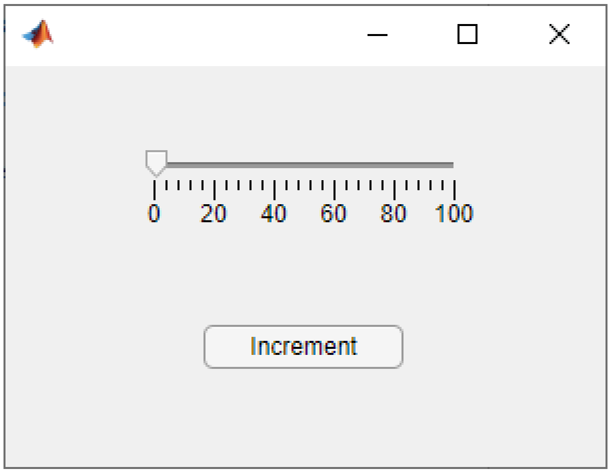
To specify an anonymous function, create a function handle with the two required source and event input arguments that executes your callback function, script, or statement.

For example, the `changeSlider` function creates a slider UI component and a button to increment the slider value. The `incrementSlider` function does not have the source and event input arguments, since it is designed to be callable either inside or outside of a callback. To execute `incrementSlider` when the button is pressed, create an anonymous function that accepts the `src` and `event` input arguments, ignores them, and executes `incrementSlider`.

To run the `changeSlider` function, save the code below to a file named `changeSlider.m` on the MATLAB path.

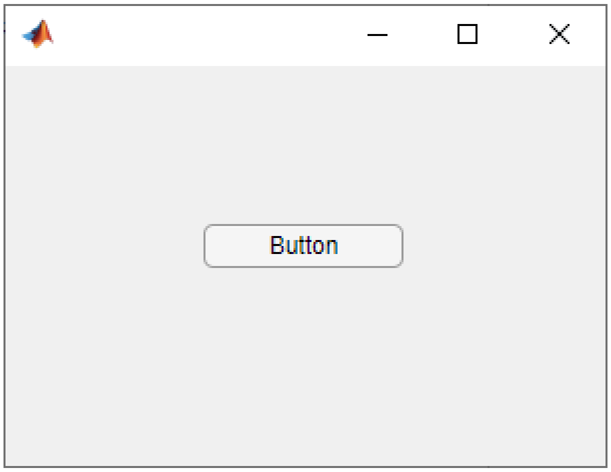
```
function changeSlider
  fig = uifigure('Position',[500 500 300 200]);
  s = uislider(fig,'Position',[75 150 150 3]);
  incrementSlider;
  b = uibutton(fig,'Position',[100 50 100 22], ...
    'Text','Increment', ...
    'ButtonPushedFcn',@(src,event)incrementSlider);

  function incrementSlider
    if s.Value < s.Limits(2)
```

```
      s.Value = s.Value + 1;
    end
  end
end
```



When your callback is a single executable statement, you can specify the callback as an anonymous function to avoid needing to define a separate function for the statement. For example, the following code creates a button that displays `Button pressed` when the button is clicked by specifying a callback as an anonymous function.

```
fig = uifigure('Position',[500 500 300 200]);
btn = uibutton(fig,'ButtonPushedFcn',@(src,event)disp('Button pressed'));
```



Unlike with callbacks specified as function handles or cell arrays, MATLAB does not check callbacks specified as anonymous functions for syntax errors and missing dependencies when you assign them to a component. If there is a problem with the anonymous function, it remains undetected until the user triggers the callback.

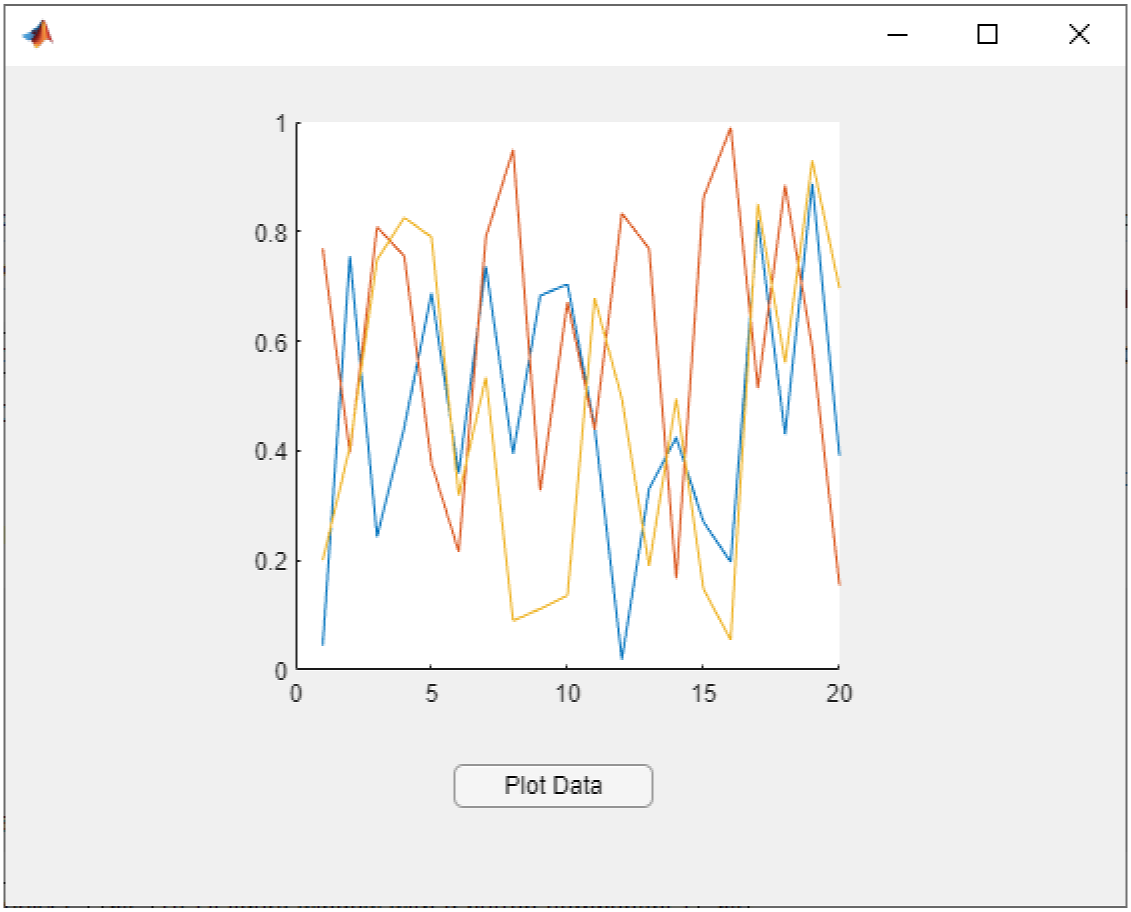
Specify Text Containing MATLAB Commands (Not Recommended)

You can specify a callback as a character vector or a string scalar when you want to execute a few simple commands, but the callback can become difficult to manage if it contains more than a few commands. Unlike with callbacks that are specified as function handles or cell arrays, MATLAB does *not* check character vectors or strings for syntax errors or missing dependencies. If there is a problem with the MATLAB expression, it remains undetected until the user triggers the callback. The character vector or string you specify must consist of valid MATLAB expressions, which can include arguments to functions.

For example, the code below creates a `UIAxes` object and a button that plots random data on the axes when it is clicked. Notice that the character vector `'plot(ax,rand(20,3))'` contains a variable, `ax` The variable `ax` must exist in the base workspace when the user triggers the callback, or MATLAB returns an error. The variable does not need to exist at the time you assign callback property value, but it must exist when the user triggers the callback.

Run the code, then click the button. Since `ax` exists in your base workspace, the callback command is valid, and MATLAB plots the data.

```
fig = uifigure;
ax = uiaxes(fig,'Position',[125 100 300 300]);
b = uibutton(fig,'Position',[225 50 100 22], ...
  'Text','Plot Data', ...
  'ButtonPushedFcn','plot(ax,rand(20,3))');
```



See Also

Related Examples

- “Share Data Among Callbacks” on page 11-9
- “Interrupt Callback Execution” on page 11-15
- “Anonymous Functions”
- “Callbacks in App Designer” on page 6-16

Share Data Among Callbacks

You can write callback functions for UI components in your app to specify how it behaves when a user interacts with it. (For more information on callback functions in apps, see “Create Callbacks for Apps Created Programmatically” on page 11-2.)

In apps with multiple interdependent UI components, the callback functions often must access data defined inside the main app function, or share data with other callbacks. For instance, if you create an app with a list box, you might want your app to update an image based on the list box option the app user chooses. Since each callback function has its own scope, you must explicitly share information about the list box options and images with those parts of your app that need to access it. To accomplish this, use your main app function to store that information in a way that can be shared with callbacks. Then, access or modify the information from within the callback functions.

Store App Data

The UI components in your app contain useful information in their properties. For example, you can find the current position of a slider by querying its `Value` property. When you create a UI component, store the component as a variable so that you can set and access its properties throughout your app code.

In addition to their pre-defined properties, all components have a `UserData` property, which you can use to store any MATLAB data. `UserData` holds only one variable at a time, but you can store multiple values as a structure array or a cell array. You can use `UserData` to store handles to the UI components in your app, as well as other app data that might need to be updated from within your app code. One useful approach is to store all your app data in the `UserData` property of the main app figure window. If you have access to any component in the app, you can access the main figure window by using the `ancestor` function. Therefore, this keeps all your app data in a location that is accessible from within every component callback.

For example, this code creates a figure with a date picker component. It stores both the date picker and today's date as a structure array in the `UserData` property of the figure.

```
fig = uifigure;
d = uidatepicker(fig);
date = datetime("today");
fig.UserData = struct("Datepicker",d,"Today",date);
```

Note Use the `UserData` property to store only the data directly related to your app user interface. If your app uses large data sets, or data that is not created or modified inside your app code, instead store this data in a separate file and access the file from within your app.

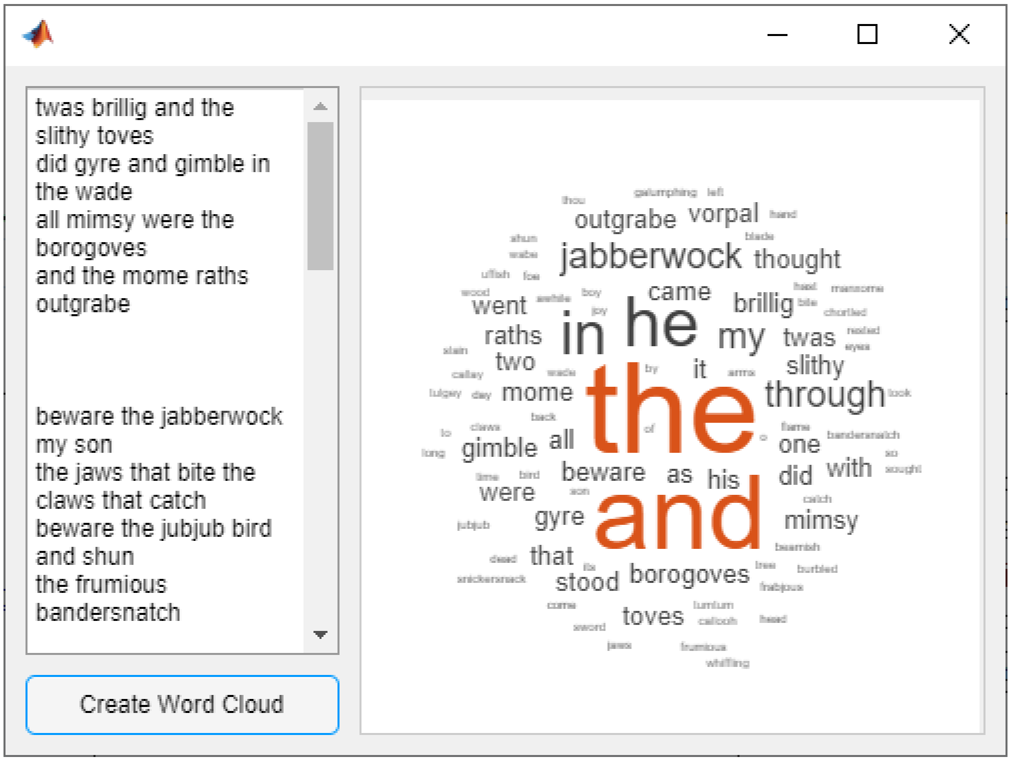
In simple applications, instead of storing your app data in the `UserData` property, you can store data as variables in your main app function, and then provide each callback with the relevant data using input arguments or nested functions.

Access App Data From Callback Functions

To access app data in a component callback function, use one of these methods:

- “Access Data in UserData” on page 11-10 — Use this method to update app data from within the callback function. It requires you to have stored app data in the `UserData` property, as described in the previous section.
- “Pass Input Data to Callbacks” on page 11-12 — Use this method in simple apps to limit what data the callback has access to, and to make it easier to reuse the callback code.
- “Create Nested Callback Functions” on page 11-13 — Use this method in simple apps to provide your callback functions with access to all the app data, and to organize your app code within a single file.

Each section below describes one of these methods, and provides an example of using the method to share data within an app. For each example, the final app behavior is the same: the app user can enter text into a text area and click a button to generate a word cloud from the text. To accomplish this, the app must share data between the text area, the button, and the panel that holds the word cloud. Each example shares this data in a different way.



twas brillig and the slithy toves
did gyre and gimble in the wade
all mimsy were the borogoves
and the mome raths outgrabe

beware the jabberwock my son
the jaws that bite the claws that catch
beware the jubjub bird and shun
the frumious bandersnatch

Access Data in UserData

To keep all your app data organized in one place, store the data somewhere that every component can easily access. First, in the setup portion of your app code, use the `UserData` property of the figure window to store any data a component needs access to from its callbacks. Since every UI component is a child of the main figure, you can access the figure from within the callback by using the `ancestor` function. For example, if your figure contains a panel with a button that is stored in a variable named `btn`, you can access the figure with this code.

```
fig = ancestor(btn,"figure","toplevel");
```

Then, once you have access to the figure from within the callback, you can access and modify the app data stored in the `UserData` of the figure.

Example: Word Cloud App Using UserData

In the word cloud app, to share app data when the app user clicks the button, store the data in the UserData property of the figure. Define a ButtonPushedFcn callback function named createWordCloud that plots a word cloud based on the text in the text area. The createWordCloud function needs access to the value of the text box at the time the button is clicked. It also needs access to the panel container to plot the data in. To provide this access, set the UserData of the figure to a struct that stores the text area component and the panel container.

```
fig.UserData = struct("TextArea",txt,"Panel",pnl);
```

In the createWordCloud function, access the UserData property of the figure. Since MATLAB automatically passes the component executing the callback to the callback function as src, you can access the figure from within the callback by using the ancestor function.

```
fig = ancestor(src,"figure","toplevel");
```

Then, you can use the figure to access the panel and the text.

```
data = fig.UserData;
txt = data.TextArea;
pnl = data.Panel;
val = txt.Value;
```

To run this example, save the shareUserData function to a file named shareUserData.m on the MATLAB path.

```
function shareUserData
  % Create figure and grid layout
  fig = uifigure;
  gl = uigridlayout(fig,[2,2]);
  gl.RowHeight = {'1x',30};
  gl.ColumnWidth = {'1x','2x'};

  % Create and lay out text area
  txt = uitextarea(gl);
  txt.Layout.Row = 1;
  txt.Layout.Column = 1;

  % Create and lay out button
  btn = uibutton(gl);
  btn.Layout.Row = 2;
  btn.Layout.Column = 1;
  btn.Text = "Create Word Cloud";

  % Create and lay out panel
  pnl = uipanel(gl);
  pnl.Layout.Row = [1 2];
  pnl.Layout.Column = 2;

  % Store data in figure
  fig.UserData = struct("TextArea",txt,"Panel",pnl);

  % Assign button callback function
  btn.ButtonPushedFcn = @createWordCloud;
end

% Process and plot text
```

```
function createWordCloud(src,event)
  fig = ancestor(src,"figure","toplevel");
  data = fig.UserData;
  txt = data.TextArea;
  pnl = data.Panel;
  val = txt.Value;

  words = {};
  for k = 1:length(val)
      text = strsplit(val{k});
      words = [words text];
  end
  c = categorical(words);
  wordcloud(pnl,c);
end
```

Pass Input Data to Callbacks

When a callback function needs access to data, you can pass that data directly to the callback as an input. In addition to the `src` and `event` inputs that MATLAB automatically passes to every callback function, you can declare your callback function with additional input arguments. Pass these inputs arguments to the callback function using a cell array or an anonymous function.

Example: Word Cloud App Using Callback Input Arguments

In the word cloud app, to share app data when the app user pushes the button, pass that data to the `ButtonPushedFcn` callback function.

Define a `ButtonPushedFcn` callback function named `createWordCloud` that plots a word cloud based on the text in the text area. The `createWordCloud` function needs access to the value of the text box at the time the button is clicked. It also needs access to the panel container to plot the data in. To provide this access, define `createWordCloud` to take the text area and panel as input arguments, in addition to the required `src` and `event` arguments.

```
function createWordCloud(src,event,txt,pnl)
  % Code to plot the word cloud
end
```

Assign the `createWordCloud` callback function and pass in the text area and panel by specifying `ButtonPushedFcn` as a cell array containing a handle to `createWordCloud`, followed by the additional input arguments.

```
btn.ButtonPushedFcn = {@createWordCloud,txt,pnl};
```

To run this example, save the `shareAsInput` function to a file named `shareAsInput.m` on the MATLAB path.

```
function shareAsInput
  % Create figure and grid layout
  fig = uifigure;
  gl = uigridlayout(fig,[2,2]);
  gl.RowHeight = {'1x',30};
  gl.ColumnWidth = {'1x','2x'};

  % Create and lay out text area
  txt = uitextarea(gl);
  txt.Layout.Row = 1;
  txt.Layout.Column = 1;

  % Create and lay out button
  btn = uibutton(gl);
  btn.Layout.Row = 2;
  btn.Layout.Column = 1;
  btn.Text = "Create Word Cloud";

  % Create and lay out panel
  pnl = uipanel(gl);
  pnl.Layout.Row = [1 2];
  pnl.Layout.Column = 2;

  % Assign button callback function
  btn.ButtonPushedFcn = {@createWordCloud,txt,pnl};
end

% Process and plot text
function createWordCloud(src,event,txt,pnl)
  val = txt.Value;
  words = {};
  for k = 1:length(val)
      text = strsplit(val{k});
      words = [words text];
  end
  c = categorical(words);
  wordcloud(pnl,c);
end
```

Create Nested Callback Functions

Finally, you can nest callback functions inside the main function of a programmatic app. When you do this, the nested callback functions share a workspace with the main function. As a result, the nested functions have access to all the UI components and variables defined in the main function.

Example: Word Cloud App Using Nested Callback

In the word cloud app, to share app data when the app user pushes the button, nest the button callback function inside the main app function. Define a `ButtonPushedFcn` callback function named `createWordCloud` that plots a word cloud based on the text in the text area. The `createWordCloud` function needs access to the value of the text box at the time the button is clicked. It also needs access to the panel container to plot the data in. To provide this access, define `createWordCloud` inside the main `nestCallback` function. The nested function has access to the `t` and `p` variables that store the text area and panel components.

To run this example, save the `nestCallback` function to a file named `nestCallback.m`, and then run it.

```
function nestCallback
  % Create figure and grid layout
  fig = uifigure;
  gl = uigridlayout(fig,[2,2]);
  gl.RowHeight = {'1x',30};
  gl.ColumnWidth = {'1x','2x'};
  % Create and lay out text area
  t = uitextarea(gl);
  t.Layout.Row = 1;
  t.Layout.Column = 1;

  % Create and lay out button
  b = uibutton(gl);
  b.Layout.Row = 2;
  b.Layout.Column = 1;
  b.Text = "Create Word Cloud";

  % Create and lay out panel
  p = uipanel(gl);
  p.Layout.Row = [1 2];
  p.Layout.Column = 2;

  % Assign button callback function
  b.ButtonPushedFcn = @createWordCloud;

  % Process and plot text
  function createWordCloud(src,event)
    val = t.Value;
    words = {};
    for k = 1:length(val)
        text = strsplit(val{k});
        words = [words text];
    end
    c = categorical(words);
    wordcloud(p,c);
  end

end
```

See Also

Related Examples

- “Nested Functions”
- “Interrupt Callback Execution” on page 11-15
- “Create Callbacks for Apps Created Programmatically” on page 11-2
- “Callbacks in App Designer” on page 6-16
- “Share Data Within App Designer Apps” on page 6-26

Interrupt Callback Execution

MATLAB lets you control whether a callback function can be interrupted while it is executing. At times you might want to permit interruptions. For instance, you might allow users to stop an animation loop by creating a callback that interrupts the animation. At other times, when the order of the running callback is important, you might want to prevent potential interruptions. For instance, in order to make an app more responsive, you might prevent interruption of callbacks that respond to pointer movement.

Interrupted Callback Behavior

Callback functions execute according to their order in a queue. If a callback is executing and a user action triggers a second callback, the second callback attempts to interrupt the first callback. The first callback is the running callback. The second callback is the interrupting callback.

Certain commands that occur in the running callback cause MATLAB to process the rest of the callback queue. MATLAB determines callback interruption behavior whenever it executes one of these commands. These commands include `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, and `pause`.

If the running callback does not contain one of these commands, then no interruption occurs. MATLAB first finishes executing the running callback, and later executes the interrupting callback.

If the running callback does contain one of these commands, then the `Interruptible` property of the object that owns the running callback determines whether the interruption occurs:

- If the value of `Interruptible` is `'on'`, then the interruption occurs. When MATLAB processes the callback queue, it pauses the execution of the running callback and executes the interrupting callback. After the interrupting callback is complete, MATLAB then resumes executing the running callback.
- If the value of `Interruptible` is `'off'`, then no interruption occurs. Instead, the `BusyAction` property of the interrupting callback determines what MATLAB does with the interrupting callback:
  - If the value of `BusyAction` is `'queue'`, MATLAB executes the interrupting callback after the running callback finishes.
  - If the value of `BusyAction` is `'cancel'`, MATLAB discards the interrupting callback.

The default value of `Interruptible` is `'on'`, and the default value of `BusyAction` is `'queue'`.

Finally, if the interrupting callback is a `DeleteFcn`, `CloseRequestFcn`, or `SizeChangedFcn` callback, then the interruption occurs regardless of the value of the `Interruptible` property.

Control Callback Interruption Behavior

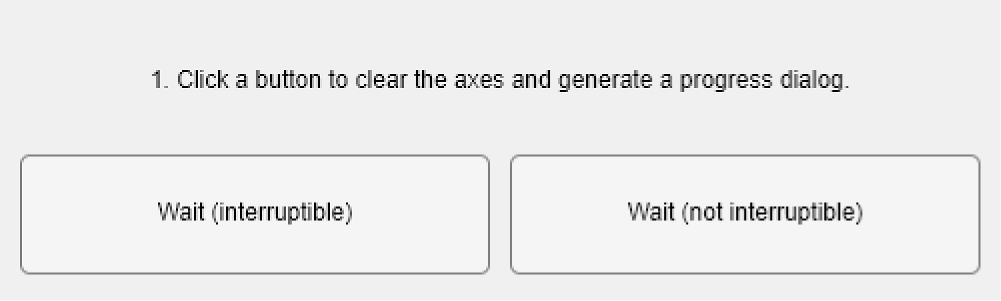
This example shows how the `Interruptible` and `BusyAction` component properties interact to produce different types of callback interruption behavior.

Create a file called `callbackBehavior.m` in your current folder, and define in it a function with the same name. This function creates an app with two figure windows, each with two buttons. Each of the buttons has a `ButtonPushedFcn` callback and a different callback execution property value. If you click one button, and then click a second button before the first one is done, then the callback of the second button attempts to interrupt the first. The buttons in the first window display and update a progress dialog when clicked. The buttons in the second window plot data when clicked. You can control what happens by defining the interruption behavior for the two buttons.

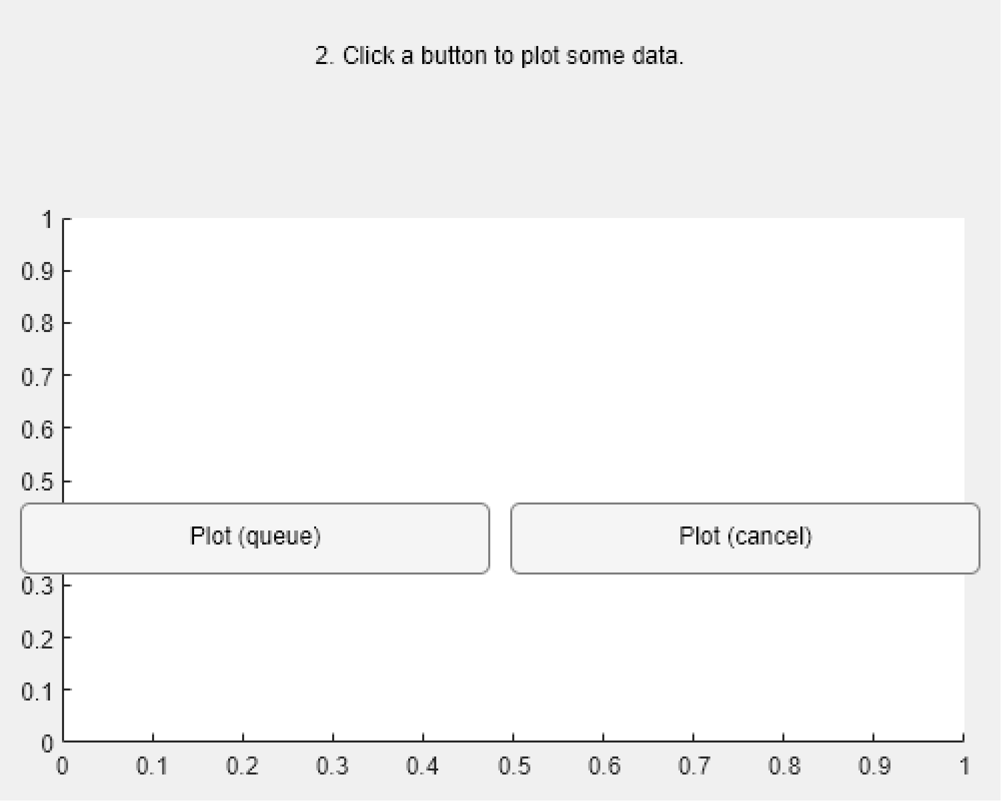
```
function callbackBehavior
% Create the figures and grid layouts
fig1 = uifigure('Position',[400 600 500 150]);
g1 = uigridlayout(fig1,[2,2]);
fig2 = uifigure('Position',[400 100 500 400]);
g2 = uigridlayout(fig2,[3,2], ...
    'RowHeight', {'1x','1x','8x'});

% Create the label for the first figure window
lbl1 = uilabel(g1,'Text','1. Click a button to clear the axes and generate a progress dialog.');
lbl1.Layout.Column = [1 2];
lbl1.HorizontalAlignment = 'center';

% Create the buttons that create a progress dialog
interrupt = uibutton(g1, ...
    'Text','Wait (interruptible)', ...
    'Interruptible','on', ...
    'ButtonPushedFcn',@createProgressDlg);
nointerrupt = uibutton(g1, ...
    'Text','Wait (not interruptible)', ...
    'Interruptible','off', ...
    'ButtonPushedFcn',@createProgressDlg);

% Create the label for the second figure window
lbl2 = uilabel(g2,'Text','2. Click a button to plot some data.');
lbl2.Layout.Column = [1 2];
lbl2.HorizontalAlignment = 'center';

% Create the axes
ax = uiaxes(g2);
ax.Layout.Row = 3;
ax.Layout.Column = [1 2];

% Create the buttons to plot data
queue = uibutton(g2, ...
    'Text','Plot (queue)', ...
    'BusyAction','queue', ...
    'ButtonPushedFcn',@(src,event)surf(ax,peaks(35)));
queue.Layout.Row = 2;
queue.Layout.Column = 1;

cancel = uibutton(g2, ...
    'Text','Plot (cancel)', ...
    'BusyAction','cancel', ...
    'ButtonPushedFcn',@(src,event)surf(ax,peaks(35)));
cancel.Layout.Row = 2;
cancel.Layout.Column = 2;

    % Callback function to create and update a progress dialog
    function createProgressDlg(src,event)
        % Clear axes
        cla(ax,'reset')
        % Create the dialog
        dlg = uiprogressdlg(fig1,'Title','Please Wait',...
        'Message','Loading...');
        steps = 250;
        for step = 1:steps
            % Update progress
            dlg.Value = step/steps;
            pause(0.01)
        end
        close(dlg)
    end
end
```

Call the `callbackBehavior` function to display the figure windows.

```
callbackBehavior
```



1. Click a button to clear the axes and generate a progress dialog.



2. Click a button to plot some data.

Click pairs of buttons to see the effects of different combinations of `Interruptible` and `BusyAction` property values.

- Callback interruption — Click **Wait (interruptible)** immediately followed by either button in the second window: **Plot (queue)** or **Plot (cancel)**. Because the first button has its `Interruptible` value set to `'on'`, interruption occurs. The plot appears while the progress dialog is still running.
- Callback queueing — Click **Wait (not interruptible)** immediately followed by **Plot (queue)**. Because the first button has its `Interruptible` value set to `'off'` and the second button has its `BusyAction` value set to `'queue'`, queueing occurs. The progress dialog runs to completion. Then, the plot displays.
- Callback cancellation — Click **Wait (not interruptible)** immediately followed by **Plot (cancel)**. Because the first button has its `Interruptible` value set to `'off'` and the second button has its `BusyAction` value set to `'cancel'`, cancellation occurs. The progress dialog runs to completion. But then, no plot appears, because MATLAB has discarded the plot callback.

See Also

`timer` | `drawnow` | `waitfor` | `uiwait`

Related Examples

- “Create Callbacks for Apps Created Programmatically” on page 11-2
- “Schedule Command Execution Using Timer”
- “Create Responsive Apps” on page 8-8

Developing Classes of UI Component Objects

Develop Custom UI Components Programmatically

To create custom UIs and visualizations, you can combine multiple graphics and UI objects, change their properties, or call additional functions. In R2020a and earlier releases, a common way to store your customization code and share it with others is to write a script or a function.

Starting in R2020b, instead of a script or function, you can create a class implementation for your UI components by defining a subclass of the `ComponentContainer` base class. Creating a class has these benefits:

- Easy customization — When users want to customize an aspect of your UI component, they can set a property rather than having to modify and rerun your code. Users can modify properties at the command line or inspect them in the Property Inspector.
- Encapsulation — Organizing your code in this way allows you to hide implementation details from your users. You implement methods that perform calculations and manage the underlying graphics objects.

This topic gives an overview of the steps to create a custom UI component by defining a class programmatically. Alternatively, starting in R2022a, you can create a custom UI component interactively using App Designer. For more information about the interactive approach, see “Create a Simple Custom UI Component in App Designer” on page 13-2.

Structure of a UI Component Class

A UI component class has several required parts, and several more that are optional.

In the first line of a UI component class, specify the `matlab.ui.componentcontainer.ComponentContainer` class as the superclass. For example, the first line of a class called `ColorSelector` looks like this:

```
classdef ColorSelector < matlab.ui.componentcontainer.ComponentContainer
```

In addition to specifying the superclass, include the following components in your class definition. Some components are required, while other components are either recommended or optional.

Component	Description
Public property block on page 12-3 (recommended)	This block defines all the properties that users have access to. Together, these properties make up the user interface of your UI component.
Private property block on page 12-3 (recommended)	This block defines the underlying graphics objects and other implementation details that users cannot access. In this block, set these attribute values: • `Access = private` • `Transient` • `NonCopyable`

Component	Description
Events block on page 12-4 (optional)	This block defines the events that this UI component will trigger. In this block, set these attribute values: • `HasCallbackProperty` • `NotifyAccess = protected` When you set the `HasCallbackProperty` attribute, MATLAB creates a public property for each event in the block. The public property stores the user-provided callback to execute when the event fires.
`setup` method on page 12-5 (required)	This method sets the initial state of the UI component. It executes once when MATLAB constructs the object. Define this method in a protected `methods` block.
`update` method on page 12-5 (required)	This method updates the underlying objects in your UI component. It executes under the following conditions: • During the next `drawnow` execution after the user changes one or more property values • When an aspect of the user's graphics environment changes (such as the size) Define this method in the same protected block as the `setup` method.

Constructor Method

You do not have to write a constructor method for your class, because it inherits one from the `ComponentContainer` base class. The inherited constructor accepts optional input arguments: a parent container and any number of name-value pair arguments for setting properties on the UI component. For example, if you define a class called `ColorSelector` that has the public properties `Value` and `ValueChangedFcn`, you can create an instance of your class using this code:

```
f = uifigure;
c = ColorSelector(f,'Value',[1 1 0],'ValueChangedFcn',@(o,e)disp('Changed'))
```

If you want to provide a constructor that has a different syntax or different behavior, you can define a custom constructor method. For an example of a custom constructor, see “Write Constructors for Chart Classes”.

Public and Private Property Blocks

Divide your class properties between at least two blocks:

- A public block for storing the components of the user-facing interface
- A private block for storing the implementation details that you want to hide

The properties that go in the public block store the input values provided by the user. For example, a UI component that allows a user to pick a color value might store the color value in a public property. Since the property name-value pair arguments are optional inputs to the implicit constructor method, the recommended approach is to initialize the public properties to default values.

The properties that go in the private block store the underlying graphics objects that make up your UI component, in addition to any calculated values that you want to store. Eventually, your class will use the data in the public properties to configure the underlying objects. Set the Transient and NonCopyable attributes for the private block to avoid storing redundant information if the user copies or saves an instance of the UI component.

For example, here are the property blocks for a UI component that allows a user to pick a color value. The public property block stores the value that the user can control: the color value. The private property block stores the grid layout manager, button, and edit field objects.

```
properties
    Value {validateattributes(Value, ...
        {'double'},{'<=',1,'>=',0,'size',[1 3]})} = [1 0 0];
end

properties (Access = private,Transient,NonCopyable)
    Grid matlab.ui.container.GridLayout
    Button matlab.ui.control.Button
    EditField matlab.ui.control.EditField
end
```

Event Block

You optionally can add a third block for events that the UI component fires.

Create a public property for each event in the block by specifying the HasCallbackProperty attribute. The public property stores the user-provided callback to execute when the event fires. The name of the public property is the name of the event appended with the letters Fcn. For example, a UI component that allows a user to pick a color value might define the event ValueChanged, which generates the corresponding public property ValueChangedFcn. Use the notify method to fire the event and execute the callback in the property.

For example, here is the event block for a UI component that allows a user to pick a color value.

```
events (HasCallbackProperty, NotifyAccess = protected)
    ValueChanged
end
```

When the user picks a color value, call the notify method to fire the ValueChanged event and execute the callback in the ValueChangedFcn property.

```
function getColorFromUser(comp)
    c = uisetcolor(comp.Value);
    if (isscalar(c) && (c == 0))
        return;
    end

    % Update the Value property
    oldValue = comp.Value;
    comp.Value = c;

    % Execute user callbacks and listeners
    notify(comp,'ValueChanged');
end
```

When a user creates an instance of the UI component, they can specify a callback to execute when the color value changes using the generated public property.

```
f = uifigure;
c = ColorSelector(f,'ValueChangedFcn',@(o,e)disp('Changed'))
```

For more information about specifying callbacks to properties, see “Create Callbacks for Apps Created Programmatically” on page 11-2.

Setup Method

Define a `setup` method for your class. A `setup` method executes once when MATLAB constructs the UI component object. Any property values passed as name-value arguments to the constructor method are assigned after this method executes.

Use the `setup` method to:

- Create graphics and UI objects that make up the component.
- Store the objects as private properties on the component object.
- Lay out and configure the objects.
- Wire up the objects to do something useful within the component.

Define the `setup` method in a protected block.

Most UI object creation functions have an optional input argument for specifying the parent. When you call these functions from within a class method, you must specify the target parent. Specify the target parent as the UI component object being set up by using the class instance argument passed to the method.

For example, consider a UI component that has these properties:

- One public property called `Value`
- Three private properties called `Grid`, `Button`, and `EditField`

The `setup` method calls the `uigridlayout`, `uieditfield`, and `uibutton` functions to create the underlying graphics object for each private property, specifying the instance of the UI component (`comp`) as the target parent.

```
function setup(comp)
    % Create grid layout to manage building blocks
    comp.Grid = uigridlayout(comp,[1 2],'ColumnWidth',{'1x',22},...
        'RowHeight',{'fit'},'ColumnSpacing',2,'Padding',2);

    % Create edit field for entering color value
    comp.EditField = uieditfield(comp.Grid,'Editable',false,...
        'HorizontalAlignment','center');

    % Create button to confirm color change
    comp.Button = uibutton(comp.Grid,'Text',char(9998), ...
        'ButtonPushedFcn',@(o,e) comp.getColorFromUser());
end
```

Update Method

Define an `update` method for your class. This method executes when your UI component object needs to change its appearance in response to a change in values.

Use the `update` method to reconfigure the underlying graphics objects in your UI component based on the new values of the properties. Typically, this method does not determine which of the properties changed. It reconfigures all aspects of the underlying graphics objects that depend on the properties.

For example, consider a UI component that has these properties:

- One public property called Value
- Three private properties called Grid, Button, and EditField

The update method updates the BackgroundColor of the EditField and Button objects with the color stored in Value. The update method also updates the EditField object with a numeric representation of the color. This way, however Value is changed, the change becomes equally visible everywhere.

```
function update(comp)
    % Update edit field and button colors
    set([comp.EditField comp.Button],'BackgroundColor',comp.Value, ...
        'FontColor',comp.getContrastingColor(comp.Value));

    % Update edit field display text
    comp.EditField.Value = num2str(comp.Value,'%0.2g ');

end
```

There might be a delay between changing property values and seeing the results of those changes. The update method runs for the first time after the setup method runs and then it runs every time drawnow executes. The drawnow function automatically executes periodically, based on the state of the graphics environment in the user's MATLAB session. This periodic execution can lead to the potential delay.

Example: Color Selector UI Component

This example shows how to create a UI component for selecting a color, using the code discussed in other sections of this page. Create a class definition file named ColorSelectorComponent.m in a folder that is on the MATLAB path. Define the class by following these steps.

Step	Implementation
Derive from the ComponentContainer base class.	`classdef ColorSelector < matlab.ui.componentcontainer.ComponentContainer`

Define public properties.	`properties`
	`    Value {validateattributes(Value, ...`
	`        {'double'},{'<=',1,'>=',0,'size',[1 3]})} = [1 0 0];`
	`end`

Define public events.	`events (HasCallbackProperty, NotifyAccess = protected)`
	`    ValueChanged % ValueChangedFcn will be the generated callback property`
	`end`

Define private properties.	`properties (Access = private, Transient, NonCopyable)`
	`    Grid matlab.ui.container.GridLayout`
	`    Button matlab.ui.control.Button`
	`    EditField matlab.ui.control.EditField`
	`end`

<table>
<tr><th>Step</th><th>Implementation</th></tr>
<tr><td>Implement the <code>setup</code> method. In this case, call the <code>uigridlayout</code>, <code>uieditfield</code>, and <code>uibutton</code> functions to create <code>GridLayout</code>, <code>EditField</code>, and <code>Button</code> objects. Store those objects in the corresponding private properties.

Specify the <code>getColorFromUser</code> method as the <code>ButtonPushedFcn</code> callback that is called when the button is pressed.</td><td><pre><code>   methods (Access = protected)
        function setup(comp)
            % Grid layout to manage building blocks
            comp.Grid = uigridlayout(comp,[1,2],'ColumnWidth',{'1x',22}, ...
                'RowHeight',{'fit'},'ColumnSpacing',2,'Padding',2);

            % Edit field for value display and button to launch uisetcolor
            comp.EditField = uieditfield(comp.Grid,'Editable',false, ...
                'HorizontalAlignment','center');
            comp.Button = uibutton(comp.Grid,'Text',char(9998), ...
                'ButtonPushedFcn',@(o,e) comp.getColorFromUser());

        end
</code></pre></td></tr>
<tr><td>Implement the <code>update</code> method. In this case, update the background color of the underlying objects and the text in the edit field to show the color value.</td><td><pre><code>   function update(comp)
            % Update edit field and button colors
            set([comp.EditField comp.Button],'BackgroundColor',comp.Value, ...
                'FontColor',comp.getContrastingColor(comp.Value));

            % Update the display text
            comp.EditField.Value = num2str(comp.Value,'%0.2g ');
        end
    end
</code></pre></td></tr>
<tr><td>Wire the callbacks and other pieces together using private methods.

When the <code>getColorFromUser</code> method is triggered by a button press, call the <code>uisetcolor</code> function to open the color picker and then call the <code>notify</code> function to execute the user callback and listener functions.

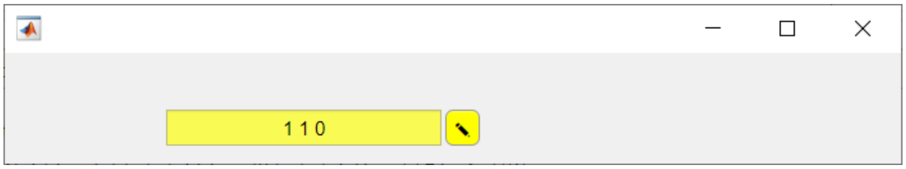
When the <code>getContrastingColo</code> <code>r</code> method is called by the <code>update</code> method, calculate whether black or white text is more readable on the new background color.</td><td><pre><code>   methods (Access = private)
        function getColorFromUser(comp)
            c = uisetcolor(comp.Value);
            if (isscalar(c) && (c == 0))
                return;
            end

            % Update the Value property
            comp.Value = c;

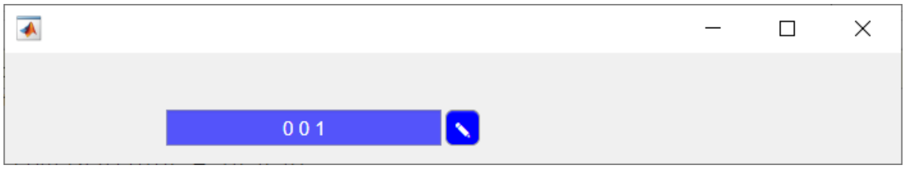
            % Execute user callbacks and listeners
            notify(comp,'ValueChanged');
        end
        function contrastColor = getContrastingColor(~,color)
            % Calculate opposite color
            c = color * 255;
            contrastColor = [1 1 1];
            if (c(1)*.299 + c(2)*.587 + c(3)*.114) > 186
                contrastColor = [0 0 0];
            end
        end
    end
end
</code></pre></td></tr>
</table>

Next, create an instance of the UI component by calling the implicit constructor method with a few of the public properties. Specify a callback to display the words `Color changed` when the color value changes.

```
h = ColorSelector('Value', [1 1 0]);
h.ValueChangedFcn = @(o,e) disp('Color changed');
```



Click the button and select a color using the color picker. The component changes appearance and MATLAB displays the words `Color changed` in the Command Window.



See Also

Classes
`matlab.ui.componentcontainer.ComponentContainer`

Functions
`uibutton` | `uieditfield` | `uigridlayout`

More About

- “Components of a Class”
- “Configure Custom UI Components for App Designer” on page 12-17

Manage Properties of Custom UI Components Programmatically

When you develop a custom UI component as a subclass of the ComponentContainer base class, you can use certain techniques to make your code more robust, efficient, and user-friendly. These techniques focus on how you define and manage the properties of your class. Use any that are helpful for the type of component you want to create and the user experience you want to provide.

- “Initialize Property Values” on page 12-9 — Set the default state of the UI component in case your users call the implicit constructor without any input arguments.
- “Validate Property Values” on page 12-9 — Ensure that the values are valid before using them.
- “Customize the Property Display” on page 12-10 — Provide a customized list of properties in the Command Window when a user references the UI component object without a semicolon.
- “Optimize the update Method” on page 12-11 — Improve the performance of the update method when only a subset of your properties are used in a time-consuming calculation.

For an example of these techniques, see “Example: Optimized Polynomial Fit UI Component with Customized Property Display” on page 12-12.

In addition, there are certain considerations and limitations to keep in mind if you want to use your custom UI component in App Designer, or share your component with users who develop apps in App Designer. These considerations are listed on a separate page, in “Configure Custom UI Components for App Designer” on page 12-17.

Initialize Property Values

Assign default values for all of the public properties of your class. This allows MATLAB to create a valid UI component even if the user omits some name-value arguments when they call the constructor method.

For UI components that contain a chart and have properties that store coordinate data, set the initial values to NaN values or empty arrays so that the default chart is empty when the user does not specify the coordinates.

Validate Property Values

Before your code uses property values, confirm that they have the correct size and class. For example, this property block validates the size and class of three properties.

```
properties
    LineColor {validateattributes(LineColor,{'double'}, ...
        {'<=',1,'>=',0,'size',[1 3]})} = [1 0 0]
    XData (1,:) double = NaN
    YData (1,:) double = NaN
end
```

LineColor must be a 1-by-3 array of class double, where each value is in the range [0,1]. Both XData and YData must be row vectors of class double.

You can also validate properties that store the underlying component objects in your UI component. To do this, you need to know the correct class name for each object. To determine the class name of an object, call the corresponding UI component function at the command line, and then call the `class` function to get the class name. For example, if you plan to create a drop-down component in your `setup` method, call the `uidropdown` function at the command line with an output argument. Then, pass the output to the `class` function to get its class name.

```
dd = uidropdown;
class(d)
```

```
ans =

    'matlab.ui.control.DropDown'
```

Use the output of the `class` function to validate the class for the corresponding property in your class. Specify the class after the property name. For example, the following property stores a `DropDown` object and validates its class.

```
properties (Access = private, Transient, NonCopyable)
        DropDown matlab.ui.control.DropDown
end
```

Occasionally, you might want to define a property that can store different shapes and classes of values. For example, if you define a property that can store a character vector, cell array of character vectors, or string array, omit the size and class validation or use a custom property validation method. For more information about validating properties, see “Validate Property Values”.

Customize the Property Display

One of the benefits of defining your UI component as a subclass of the `ComponentContainer` base class is that it also inherits from the `matlab.mixin.CustomDisplay` class. This lets you customize the list of properties MATLAB displays in the Command Window when you reference the UI component without a semicolon. To customize the property display, overload the `getPropertyGroups` method. Within that method, you can customize which properties are listed and the order of the list. For example, consider a `FitPlot` class that has the following public properties.

```
properties
    LineColor {validateattributes(LineColor,{'double'}, ...
        {'<=',1,'>=',0,'size',[1 3]})} = [1 0 0]
    XData (1,:) double = NaN
    YData (1,:) double = NaN
end
```

The following `getPropertyGroups` method specifies the scalar object property list as `XData`, `YData`, and `LineColor`.

```
function propgrp = getPropertyGroups(comp)
    if ~isscalar(comp)
        % List for array of objects
        propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(comp);
    else
        % List for scalar object
        propList = {'XData','YData','LineColor'};
        propgrp = matlab.mixin.util.PropertyGroup(propList);
    end
end
```

When the user references an instance of this UI component without a semicolon, MATLAB displays the customized list.

```
p = FitPlot
```

```
p = 

  FitPlot with properties:

    XData: NaN
    YData: NaN
    LineColor: [1 0 0]
```

For more information about customizing the property display, see “Customize Property Display”.

Optimize the update Method

In most cases, the `update` method of your class reconfigures all the relevant aspects of your UI component that depend on the public properties. Sometimes, the reconfiguration involves an expensive calculation that is time consuming. If the calculation involves only a subset of the properties, you can design your class to execute that code only when it is necessary.

One way to optimize the `update` method is to add these elements to your class:

- A private property called `ExpensivePropChanged` that accepts a `logical` value. This property indicates whether any of the properties used in the expensive calculation have changed.
- A `set` method for each property involved in the expensive calculation. Within each `set` method, set the `ExpensivePropChanged` property to `true`.
- A protected method called `doExpensiveCalculation` that performs the expensive calculation.
- A conditional statement in the `update` method that checks the value of `ExpensivePropChanged`. If the value is `true`, execute `doExpensiveCalculation`.

The following code provides a template for this design.

```
classdef OptimizedUIComponent <  matlab.ui.componentcontainer.ComponentContainer

    properties
        Prop1
        Prop2
    end
    properties(Access=private,Transient,NonCopyable)
        ExpensivePropChanged (1,1) logical = true
    end

    methods(Access = protected)
        function setup(comp)
            % Configure UI component
            % ...
        end
        function update(comp)
            % Perform expensive computation if needed
            if comp.ExpensivePropChanged
                doExpensiveCalculation(comp);
                comp.ExpensivePropChanged = false;
            end

            % Update other aspects of UI component
            % ...
        end
        function doExpensiveCalculation(comp)
            % Expensive code
            % ...
        end
    end

    methods
        function set.Prop2(comp,val)
            comp.Prop2 = val;
            comp.ExpensivePropChanged = true;
        end
    end
end
```

In this case, `Prop2` is involved in the expensive calculation. The `set.Prop2` method sets the value of `Prop2`, and then it sets `ExpensivePropChanged` to `true`. The next time the `update` method runs, it calls `doExpensiveCalculation` only if `ExpensivePropChanged` is `true`. Then, the `update` method continues to update other aspects of the UI component.

Example: Optimized Polynomial Fit UI Component with Customized Property Display

This example defines a `FitPlot` class for interactively displaying best fit polynomials, and uses all four of these best practices. The properties defined in the properties block have default values and use size and class validation. The `getPropertyGroups` method defines a custom order for the property display. The `changeFit` method performs the potentially expensive polynomial fit calculation, and the `update` method executes `changeFit` only if the plotted data changed.

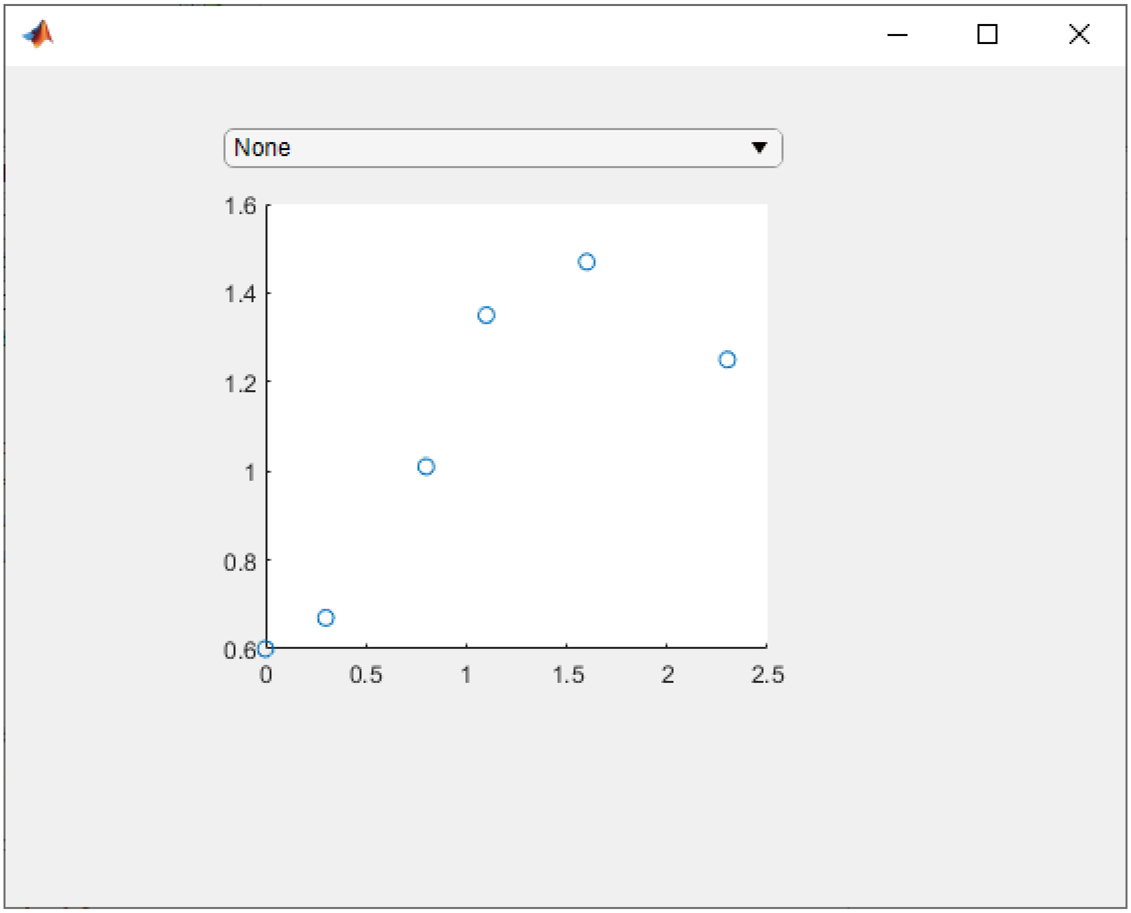
To define this class, save the `FitPlot` class definition to a file named `FitPlot.m` in a folder that is on the MATLAB path.

```
classdef FitPlot < matlab.ui.componentcontainer.ComponentContainer
    % Choose a fit method for your plotted data

    properties
        LineColor {validateattributes(LineColor,{'double'}, ...
            {'<=',1,'>=',0,'size',[1 3]})} = [1 0 0]
        XData (1,:) double = NaN
        YData (1,:) double = NaN
    end

    properties (Access = private, Transient, NonCopyable)
        DropDown matlab.ui.control.DropDown
        Axes matlab.ui.control.UIAxes
        GridLayout matlab.ui.container.GridLayout
        DataLine (1,1) matlab.graphics.chart.primitive.Line
        FitLine (1,1) matlab.graphics.chart.primitive.Line
        FitXData (1,:) double
        FitYData (1,:) double
        ExpensivePropChanged (1,1) logical = true
    end

    methods (Access=protected)
        function setup(comp)
            % Set the initial position of this component
            comp.Position = [100 100 300 300];

            % Create the grid layout, drop-down, and axes
            comp.GridLayout = uigridlayout(comp,[2,1], ...
                'RowHeight',{20,'1x'},...
                'ColumnWidth',{'1x'});
            comp.DropDown = uidropdown(comp.GridLayout, ...
                'Items',{'None','Linear','Quadratic','Cubic'}, ...
                'ValueChangedFcn',@(s,e) changeFit(comp));
            comp.Axes = uiaxes(comp.GridLayout);

            % Create the line objects
            comp.DataLine = plot(comp.Axes,NaN,NaN,'o');
            hold(comp.Axes,'on');
            comp.FitLine = plot(comp.Axes,NaN,NaN);
            hold(comp.Axes,'off');
        end

        function update(comp)
            % Update data points
            comp.DataLine.XData = comp.XData;
            comp.DataLine.YData = comp.YData;

            % Do an expensive operation
            if comp.ExpensivePropChanged
                comp.changeFit();
                comp.ExpensivePropChanged = false;
            end

            % Update the fit line
            comp.FitLine.Color = comp.LineColor;
            comp.FitLine.XData = comp.FitXData;
            comp.FitLine.YData = comp.FitYData;
        end

        function changeFit(comp)
            % Calculate the fit line based on the drop-down value
            if strcmp(comp.DropDown.Value,'None')
                comp.FitXData = NaN;
                comp.FitYData = NaN;
            else
                switch comp.DropDown.Value
                    case 'Linear'
                        f = polyfit(comp.XData,comp.YData,1);
                    case 'Quadratic'
                        f = polyfit(comp.XData,comp.YData,2);
                    case 'Cubic'
                        f = polyfit(comp.XData,comp.YData,3);
                end
                comp.FitXData = linspace(min(comp.XData),max(comp.XData));
                comp.FitYData = polyval(f,comp.FitXData);
            end
        end

        function propgrp = getPropertyGroups(comp)
            if ~isscalar(comp)
                % List for array of objects
                propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(comp);
            else
                % List for scalar object
                propList = {'XData','YData','LineColor'};
                propgrp = matlab.mixin.util.PropertyGroup(propList);
            end
        end

    end

    methods
        function set.XData(comp,val)
            comp.XData = val;
            comp.ExpensivePropChanged = true;
        end
        function set.YData(comp,val)
            comp.YData = val;
            comp.ExpensivePropChanged = true;
        end
    end
end
```

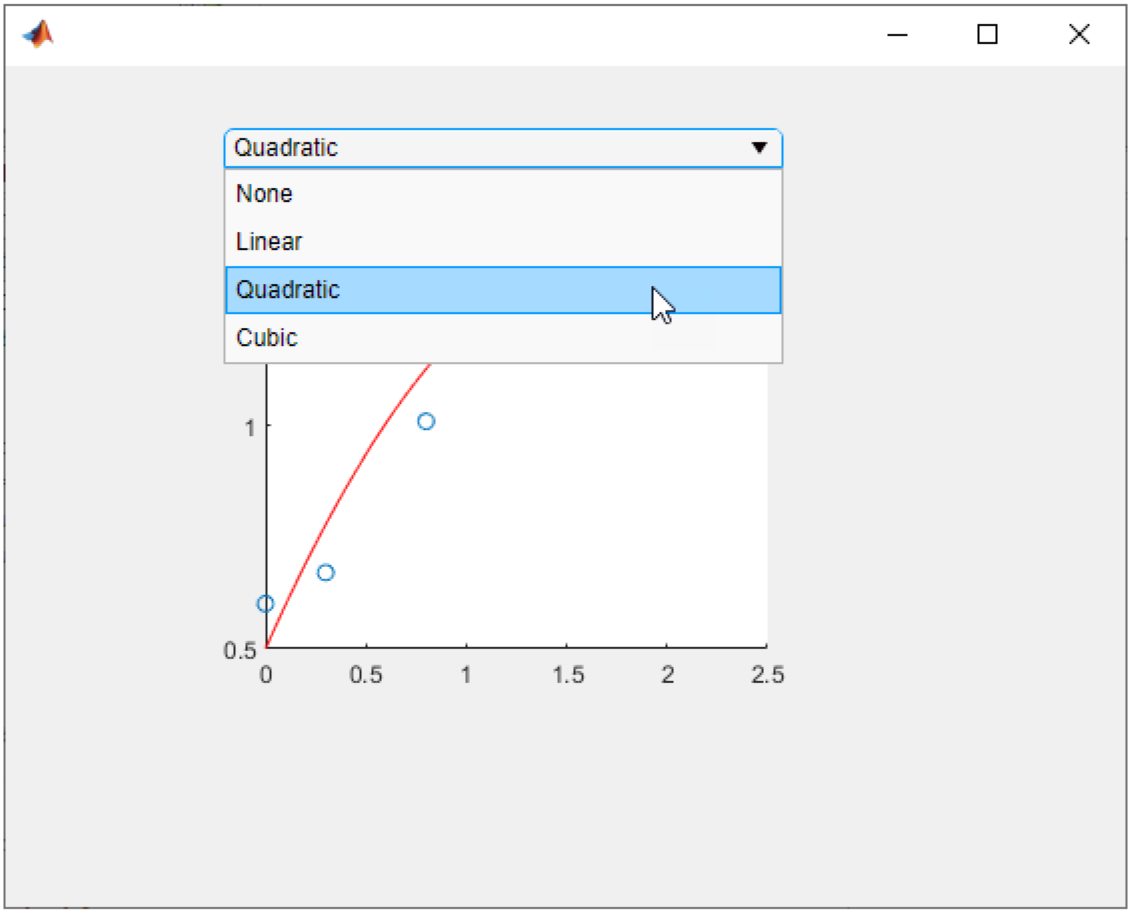
Define some sample data and use it to create an instance of `FitPlot`.

```
x = [0 0.3 0.8 1.1 1.6 2.3];
y = [0.6 0.67 1.01 1.35 1.47 1.25];
p = FitPlot('XData',x,'YData',y)
```

```
ans = 

  FitPlot with properties:

        XData: [1×43 double]
        YData: [1×43 double]
    LineColor: [1 0 0]
```

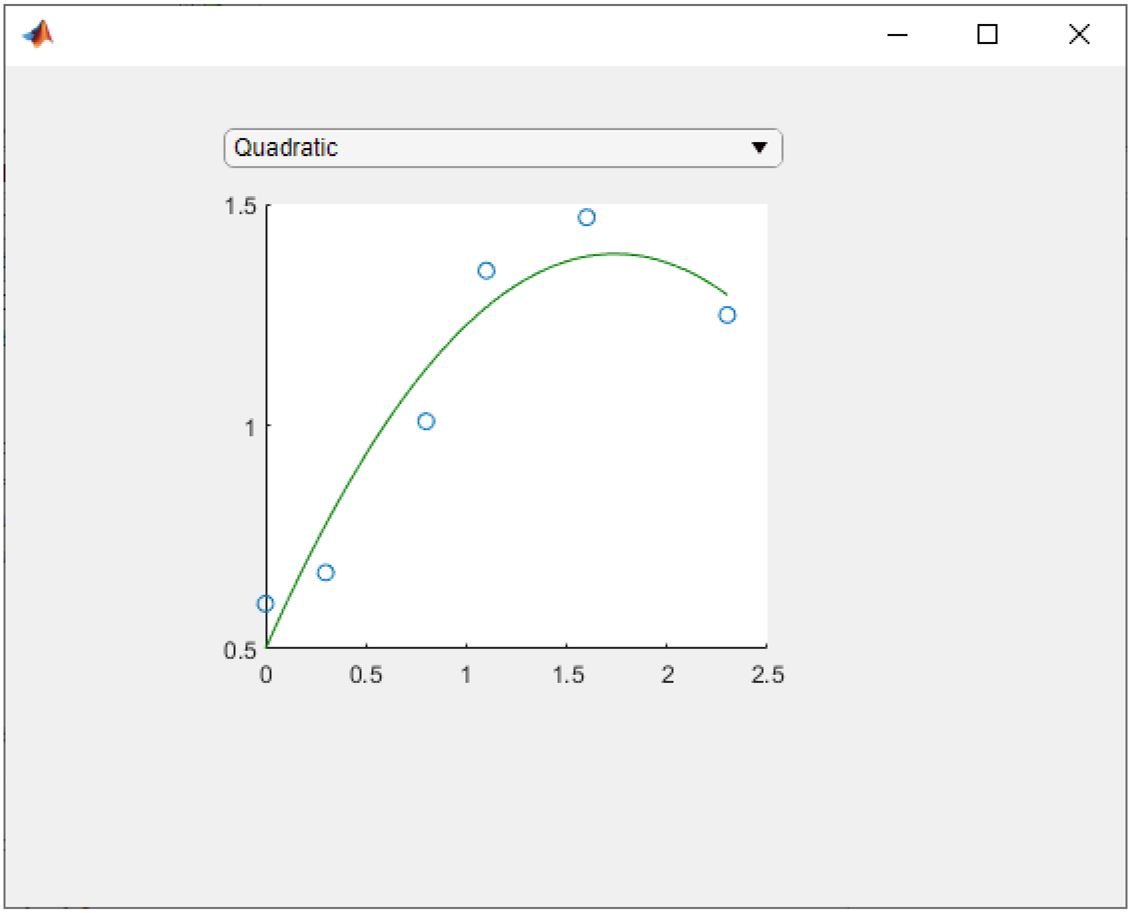


Use the drop-down to display the quadratic best fit curve.



Set the `LineColor` property to change the color of the best fit curve to green.

```
p.LineColor = [0 0.5 0];
```



See Also

Classes
`matlab.ui.componentcontainer.ComponentContainer`

Functions
`uidropdown` | `uigridlayout` | `polyfit`

More About

- “Validate Property Values”
- “Customize Property Display”
- “Property Get and Set Methods”
- “Develop Custom UI Components Programmatically” on page 12-2

Configure Custom UI Components for App Designer

In this section...

Starting in R2021a, when you create a custom UI component, you can configure your component for app creators to use interactively in App Designer. After you configure your UI component, app creators can add the component to the **Component Library** and can interact with the component on the App Designer canvas and in the Property Inspector.

Follow these configuration steps if you have created a custom UI component, either interactively in App Designer or programmatically as a subclass of the `matlab.ui.componentcontainer.ComponentContainer` base class, and you would like to use it in either of these ways:

- Access your UI component from the App Designer **Component Library** and interactively use it to create an App Designer app.
- Share your UI component for others to use interactively to create apps in App Designer.

For more information about creating a custom UI component, see:

- "Create a Simple Custom UI Component in App Designer" on page 13-2 to create a component interactively
- "Develop Custom UI Components Programmatically" on page 12-2 to create a component programmatically

Custom UI Component Prerequisites

To allow your custom UI component to be used interactively in App Designer, there are some requirements that your UI component class must satisfy.

To successfully configure your UI component, the `setup` method of your UI component class cannot have required input arguments. Also, the component class cannot dynamically add additional UI components to its parent container. The only exception is that the class can dynamically add a `ContextMenu` component in the parent figure.

For a public property of your component class to appear in the Property Inspector, you must specify its type or assign a default value to it. If the property is an enumeration, you must *both* specify its type and assign it a default value. In addition, the property type must belong to the list of types supported by App Designer. This table shows the allowable property types and their appearance in the Property Inspector.

Property Category	Supported Data Types	Property Inspector Input
Numerical	Scalars or arrays of type `single`, `double`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, or `uint64`	Edit field
Logical	`logical`	Check box
Text	Scalars of type `string`, scalars or row vectors of type `char`, and scalars or vectors of type `cell`	Text area
Enumeration	`enumeration`	Editable drop-down list

For more information on specifying property types and assigning default values, see:

- “Create Public Properties for Custom UI Components in App Designer” on page 13-13 for components created interactively
- “Manage Properties of Custom UI Components Programmatically” on page 12-9 for components created programmatically

Configure Custom UI Component

The way you configure your custom UI component for use in App Designer depends on whether you created the component interactively in App Designer or programmatically as a subclass of the `ComponentContainer` base class.

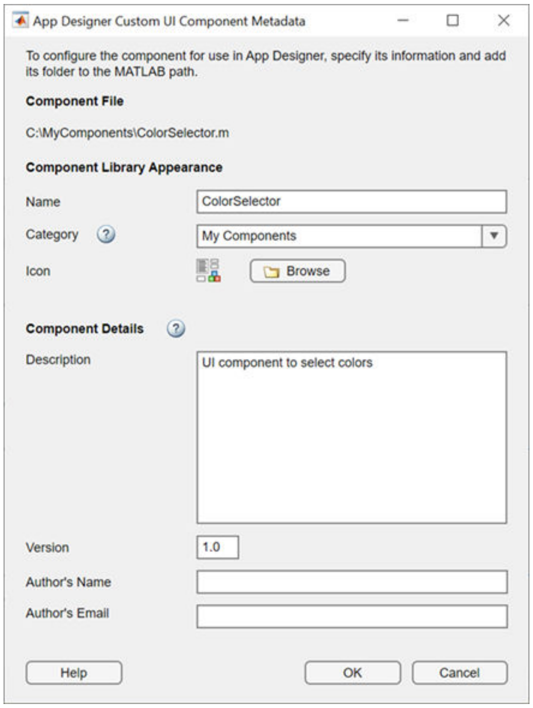
- Components created in App Designer — Open the component in App Designer. In the **File** section of the **Designer** tab, click 🛠 **Configure**.

  Alternatively, call the `appdesigner.customcomponent.configureMetadata` function from the MATLAB Command Window and pass it a path to your component MLAPP file.
- Components created programmatically — Call the `appdesigner.customcomponent.configureMetadata` function from the MATLAB Command Window and pass it a path to your component class file.

  For example, to configure a custom UI component saved as `ColorSelector.m` in the folder `C:\MyComponents`, use this command:

  ```
  appdesigner.customcomponent.configureMetadata('C:\MyComponents\ColorSelector.m');
  ```

Following these steps opens the App Designer Custom UI Component Metadata dialog box. This dialog box allows you to specify metadata about the component. App Designer uses this metadata to display the component in the **Component Library**.



App Designer Custom UI Component Metadata

To configure the component for use in App Designer, specify its information and add its folder to the MATLAB path.

Component File

C:\MyComponents\ColorSelector.m

Component Library Appearance

Name: ColorSelector

Category: My Components

Icon

Component Details

Description: UI component to select colors

Version: 1.0

Author's Name:

Author's Email:

The dialog box prepopulates all of the required metadata from the component class definition. You can edit the prepopulated metadata using the form. Select **OK** to configure the UI component.

After you select **OK**, the function creates a folder named `resources` in the same folder as your custom component file. Inside the `resources` folder, the function generates a file named `appDesigner.json`. This file contains the metadata you provided in the dialog box, in addition to other metadata MATLAB needs to make your component available in App Designer.

Note Do not modify the `appDesigner.json` file by hand. To change any custom UI component metadata, reconfigure the component.

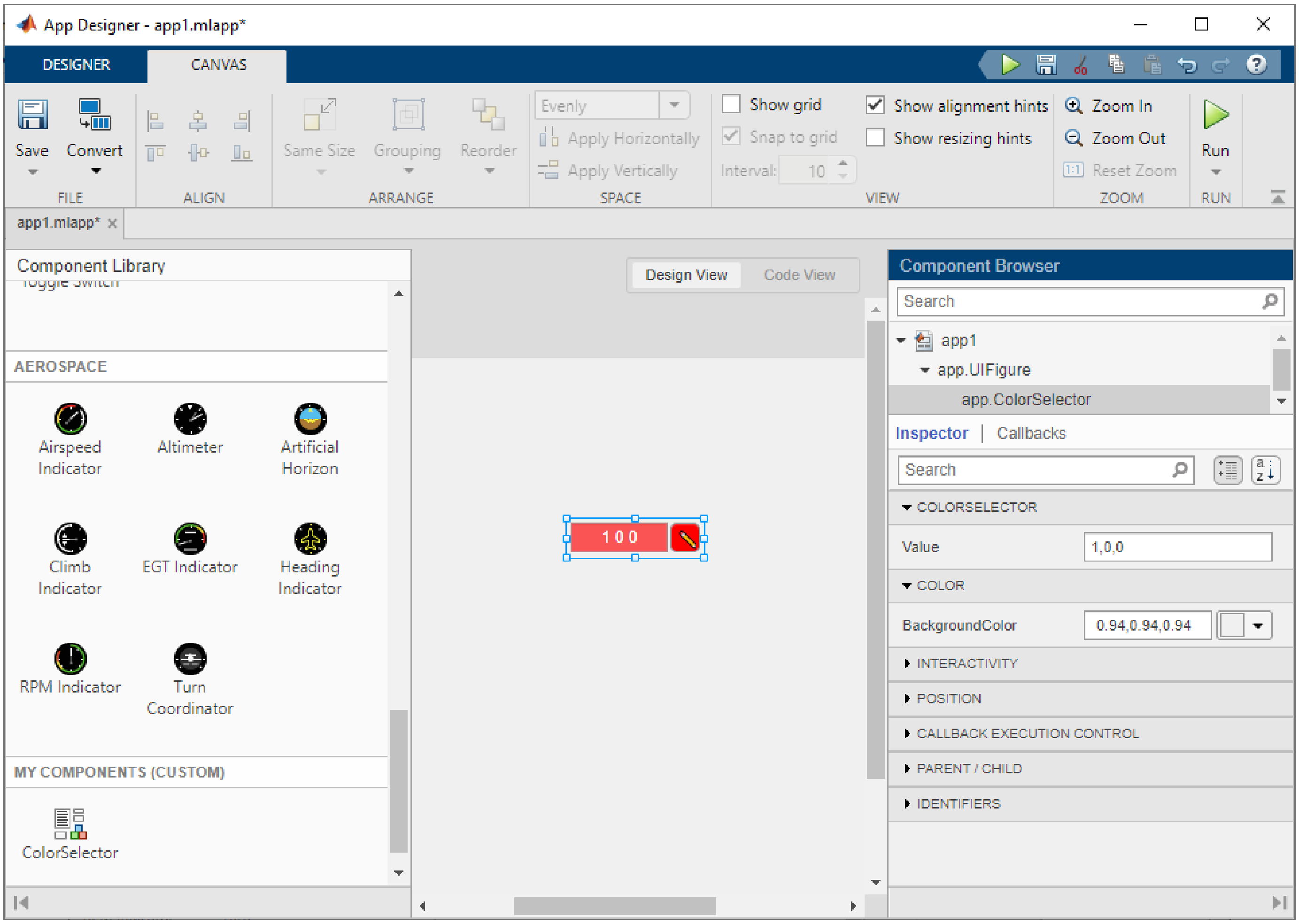
View Configured UI Component in App Designer

After you configure your custom UI component, you can view and use it in App Designer. For the UI component to appear in the App Designer **Component Library**, you must add the folder containing the component file and generated `resources` folder to the MATLAB path.

For example, if you have create a `ColorSelector` custom component, save it to a folder named `MyComponents`, and configure its App Designer metadata, follow these steps to use the component in App Designer:

1. Add the `MyComponents` folder to the MATLAB path by following the steps in “Change Folders on Search Path”.
2. Open App Designer and select **Blank App**.
3. Drag the component from the **Component Library** onto the App Designer canvas.

The Property Inspector for the component lists the public properties and callbacks of the component.



Evenly

□ Show grid
☑ Snap to grid
Interval: 10

☑ Show alignment hints
□ Show resizing hints

Value 1,0,0

BackgroundColor 0.94,0.94,0.94

Note Avoid making changes to a custom component file while using the component in an App Designer app, as doing so might lead to errors or unexpected behavior.

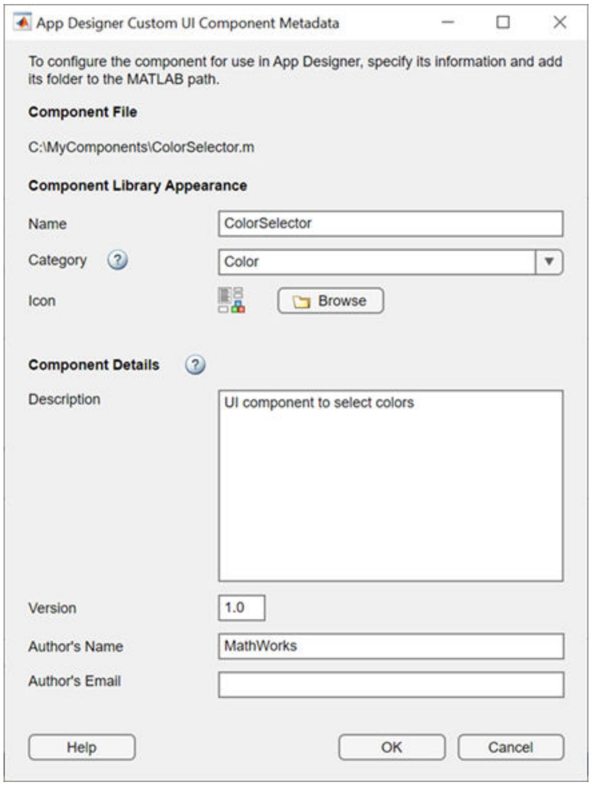
Reconfigure Custom UI Component

Reconfigure a previously configured UI component when:

- You want to change existing UI component metadata and update how the component is displayed in the App Designer **Component Library**.
- You have made changes to the UI component position or layout in your class definition.

To reconfigure your UI component, follow the same steps in the “Configure Custom UI Component” on page 12-18 section. The App Designer Custom UI Component Metadata dialog box opens with the existing metadata prepopulated.

Update the metadata, and then select **OK**.



App Designer Custom UI Component Metadata

To configure the component for use in App Designer, specify its information and add its folder to the MATLAB path.

Component File

C:\MyComponents\ColorSelector.m

Component Library Appearance

Name: ColorSelector

Category: Color

Icon

Component Details

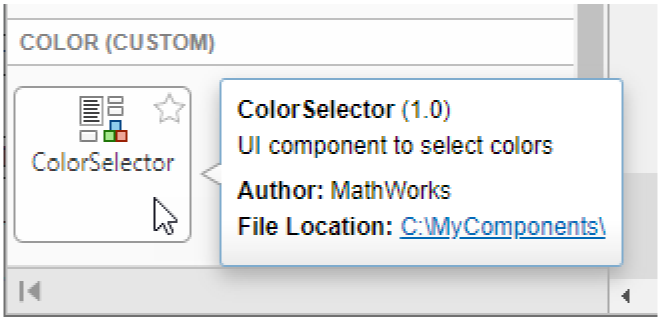
Description: UI component to select colors

Version: 1.0

Author's Name: MathWorks

Author's Email:

Go back to App Designer. The component appears in the **Component Library** with the updated configuration options.



COLOR (CUSTOM)

ColorSelector (1.0)
UI component to select colors
Author: MathWorks
File Location: C:\MyComponents\

Remove UI Component from App Designer

To remove a custom UI component from the App Designer **Component Library**, use the `appdesigner.customcomponent.removeMetadata` function.

Call the function by passing it the path to your component file. The function removes the metadata for the UI component from the `appDesigner.json` file inside the `resources` folder and removes the component from the App Designer **Component Library**.

```
appdesigner.customcomponent.removeMetadata('C:\MyComponents\ColorSelector.m');
```

After you remove the App Designer metadata for a custom UI component, any App Designer apps that use it do not load correctly. To continue editing an app that uses the UI component, reconfigure the component before you open the app.

Share Configured UI Component

After configuring a UI component, you can share the component for others to use in App Designer. You can either share the relevant files directly or package the component as a toolbox. In either case, you must also share the generated `resources` folder.

Share UI Component Files Directly

To share a configured UI component directly with a user, create and share a folder with these contents:

- The UI component class file
- The generated `resources` folder

Instruct the user you are sharing the UI component with to add the shared folder to the MATLAB path.

Package UI Component as a Toolbox

Package your UI component as a toolbox by following the steps in “Create and Share Toolboxes”. Make sure the folder you package as a toolbox has these contents:

- The UI component class file
- The generated `resources` folder

You can share the resulting `.mltbx` file directly with your users. To install it, they must double-click the `.mltbx` file in the MATLAB **Current Folder** browser.

Alternatively, you can share your UI component as an add-on by uploading the `.mltbx` file to MATLAB Central File Exchange. Your users can find and install your add-on from the MATLAB Toolstrip by performing these steps:

1. In the MATLAB Toolstrip, on the **Home** tab, in the **Environment** section, select **Add-Ons** .
2. Find the add-on by browsing through available categories on the left side of the Add-On Explorer window. Alternatively, use the search bar to search for an add-on using a keyword.
3. Click the add-on to open its detailed information page.
4. On the information page, click **Add** to install the add-on.

Troubleshoot Missing Custom UI Component

To open an app that contains a custom UI component, the component file and generated `resources` folder must be on the MATLAB path. If App Designer cannot load a custom UI component, it will display a warning dialog box when the app is opened. If you encounter this dialog box when opening an app, follow these steps to load the missing component:

1. Make sure that the UI component file and generated `resources` folder with the component metadata are in a single folder.
2. Add the folder containing the component file and `resources` folder to the MATLAB path by following the steps in “Change Folders on Search Path”.
3. Reopen the app that contains the custom UI component. The app and component should now load as expected.

See Also

Functions
`appdesigner.customcomponent.configureMetadata` |
`appdesigner.customcomponent.removeMetadata`

Classes
`matlab.ui.componentcontainer.ComponentContainer`

Related Examples

- "Create a Simple Custom UI Component in App Designer" on page 13-2
- "Develop Custom UI Components Programmatically" on page 12-2
- "Configure Property Display for Custom UI Components in App Designer" on page 13-50

Create Custom UI Component With HTML

To extend your custom UI component using third-party visualizations or widgets, create a custom component that contains an HTML UI component. Use the underlying HTML UI component to customize the component appearance and to interface with third-party libraries, and use the custom component capabilities to define component properties and callbacks that the user can set.

Custom Component Overview

To create a custom UI component that uses an HTML UI component, there are two files that you must create.

- Custom UI component file — In this file, you define your custom component. You specify its properties, its property values, the events it listens for, and the callback functions it executes.
- HTML source file — In this file, you configure and update the visual appearance of the UI component, listen for user interactions, and pass the information that an interaction has occurred to the custom UI component class.

Your code must communicate changes to property values and user interactions across these two files.

Enable Response to Property Updates

Since the custom UI component file defines the properties that users can set, but the HTML source file controls the visual style of the component, these two files need to communicate about property updates.

In the UI component file, configure the properties of your UI component. Specify the properties that users can set by defining them as public properties in a `properties` block. In the `update` method of your class, store the values of the public properties as fields in a `struct` in the `Data` property of your HTML UI component. This gives the HTML source file access to these property values.

In the HTML source file, use the property values to update the appearance of the HTML UI component. To do so, in the `setup` function inside of a `<script>` tag, access the values of the fields in `Data` and use them to modify the style properties of your HTML elements.

Enable Response to User Interactions

Users define component callback functions in MATLAB, but these callbacks often listen for a response to an action performed on an HTML element defined in the HTML source file. So these two files also need to communicate about user interactions.

In the UI component class file, first create the callback properties of your UI component. Create an `events` block with the `HasCallbackProperty`. When you define an event in this block, MATLAB creates an associated public callback property for the UI component. For example, if you create an event named `ButtonPushed`, this will automatically create a public property for your class named `ButtonPushedFcn`.

To execute a user-defined callback function associated with a user interaction, your code must first recognize when the user interaction has occurred. In the UI component class file, give the HTML UI component a way to do this. In the HTML source file, in the `setup` function inside of a `<script>` tag, create an event listener that listens for the user interaction. When the listener detects the interaction, inform MATLAB that the interaction has occurred by using the `sendEventToMATLAB` JavaScript function.

After the UI component class file receives the information that a user interaction has occurred, it must then trigger the callback associated with the interaction. To do this, in the `setup` method of the custom UI component, create an `HTMLEventReceivedFcn` callback function for the HTML UI component. This function executes whenever the component receives an event from the HTML source. In the callback function, call the `notify` function on the custom UI component event you defined. This executes the user-defined callback function associated with the event.

RoundButton Class Implementation

This example demonstrates a typical structure for writing a custom UI component that uses an HTML UI component. The example shows how to create a custom button component as a subclass of the `ComponentContainer` base class. For an example of a custom button component created in App Designer, see "Create Custom Button with Hover Effect Using HTML" on page 13-43.

The class creates a button with a custom rounded style. It allows users to specify the button color, text, text color, and response on click.

To define your UI component class, create two files in the same folder on the MATLAB path:

- `RoundButton.m` — UI component class definition
- `RoundButton.html` — HTML source file

RoundButton.m Class Definition

RoundButton class	Discussion
`classdef RoundButton < matlab.ui.componentcontainer.ComponentContainer`	Create a custom UI component named `RoundButton` by defining a subclass of the `matlab.ui.componentcontainer.ComponentContainer` class.

`properties`
`Color {mustBeMember(Color, ...`
`{'white','blue','red','green','yellow'})} = 'white'`
`FontColor {mustBeMember(FontColor, ...`
`{'black','white'})} = 'black'`
`Text (1,:) char = 'Button';`
`end`
	Define the `Color`, `FontColor`, and `Text` public properties for your `RoundButton` class. These are properties that the user can set when creating a `RoundButton` instance.
	For more information on defining properties, see "Manage Properties of Custom UI Components Programmatically" on page 12-9.

`properties (Access = private, Transient, NonCopyable)`
`HTMLComponent matlab.ui.control.HTML`
`end`
	Define the `HTMLComponent` private property to hold the HTML UI component.

`events (HasCallbackProperty, NotifyAccess = protected)`
`% Generate a ButtonPushedFcn callback property`
`ButtonPushed`
`end`
	Define a `ButtonPushed` event in an `events` block. Specify the `HasCallbackProperty` for the `events` block to automatically generate a `ButtonPushedFcn` public property for the class.

`methods (Access=protected)`	Create a `methods` block.

RoundButton class	Discussion

`        function setup(comp)`
`            % Set the initial position of this component`
`            comp.Position = [100 100 80 40];`
 	
`            % Create the HTML component`
`            comp.HTMLComponent = uihtml(comp);`
`            comp.HTMLComponent.Position = [1 1 comp.Position(3:4)];`
`            comp.HTMLComponent.HTMLSource = fullfile(pwd,"RoundButton.html");`
`            comp.HTMLComponent.HTMLEventReceivedFcn = @(src,event) notify(comp,"ButtonPushed");`
`        end`
	Define the `setup` method for your class. Within the method, set the initial position of your component relative to its parent container.
	
	Then, create an HTML component by calling the `uihtml` function. Set the following properties for your HTML component:
	• `Position` — The position of the HTML component relative to the position of the custom UI component.
	• `HTMLSource` — The source file that contains the HTML markup for the HTML component.
	• `HTMLEventReceivedFcn` — An anonymous function that sends out a notification for the `ButtonPushed` event, which executes the user-defined `ButtonPushedFcn` callback. The `HTMLEventReceivedFcn` callback function runs when the component receives an event from the HTML source.

`        function update(comp)`
`            % Update the HTML component data`
`            comp.HTMLComponent.Data.Color = comp.Color;`
`            comp.HTMLComponent.Data.FontColor = comp.FontColor;`
`            comp.HTMLComponent.Data.Text = comp.Text;`
`            comp.HTMLComponent.Position = [1 1 comp.Position(3:4)];`
`        end`
	Define the `update` method for your class. Within the method, store the values of the `Color`, `FontColor`, and `Text` properties as fields in the `Data` property of the HTML component. This enables you to update the attributes of the HTML button element, and lets the HTML component listen for when these properties are changed.

`    end`
`end`
	Close the `methods` block and the class definition.

RoundButton.html Source Definition

HTML Source	Discussion

`<!DOCTYPE html>`
`<html>`
`<head>`
	Open the `<html>` tag and the `<head>` tag.

HTML Source	Discussion

`<style>`
`html, body {`
`  height: 100%;`
`  text-align: center;`
`}`
 	
`button {`
`  width: 100%;`
`  height: 100%;`
`  border-radius: 2em;`
`  font-size: 1em;`
`  cursor: pointer;`
`  border: none;`
`}`
 	
`button:focus {`
`  outline: 0;`
`}`
`</style>`
	Define the style for the HTML content using CSS markup:
	• Set the height of the HTML body to scale to fill the entire container in which it is displayed.
	• Define the relative size of the button within the document body, the radius of the button edges, the font size, the cursor style when pointing to the button, and the button border style.

`  <script type="text/javascript">`
`    function setup(htmlComponent) {`
	Write a `setup` function inside of a `<script>` tag to connect your JavaScript object, called `htmlComponent`, to the HTML UI component you created in MATLAB.

`    htmlComponent.addEventListener("DataChanged", function(event) {`
`      buttonElement = document.getElementById("roundButton");`
`      buttonElement.style.backgroundColor = htmlComponent.Data.Color;`
`      buttonElement.innerHTML = htmlComponent.Data.Text;`
`      buttonElement.style.color = htmlComponent.Data.FontColor;`
`    });`
	Add an event listener to the `htmlComponent` JavaScript object. This event listener listens for any change in the `Data` property of the `HTMLComponent` MATLAB object, and then updates the attributes of the HTML button element in accordance with the `RoundButton` property values.

`    button = document.getElementById("roundButton");`
`    button.addEventListener("click", function(event) {`
`      htmlComponent.sendEventToMATLAB("ButtonClicked");`
`    });`
	Add an event listener to the HTML button. This event listener listens for the button element to be clicked. When a user clicks the button, the function sends an event to MATLAB to notify the MATLAB HTML object that the interaction occurred.

`    }`
`  </script>`
`</head>`
	Close the `setup` function and the `<script>` and `<head>` tags.

`  <body>`
`    <button id="roundButton"></button><br/>`
`  </body>`
	Create a button element in the body of the HTML document.

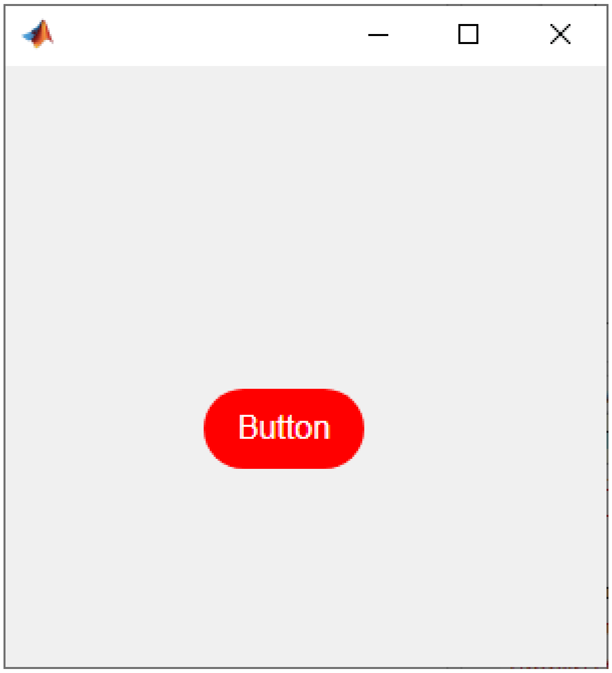
`</html>`	Close the `<html>` tag.

Create a RoundButton Instance

After creating and saving `RoundButton.m` and `RoundButton.html`, create an instance of the `RoundButton` class in a UI figure.

Specify the Color, FontColor, and the ButtonPushedFcn callback properties as name-value arguments.

```
fig = uifigure("Position",[200 200 300 300]);
btn = RoundButton(fig, ...
    "Color","red", ...
    "FontColor","white", ...
    "ButtonPushedFcn",@(src,event) disp("Clicked"));
```



Click the button. The Command Window displays Clicked.

See Also

Classes
matlab.ui.componentcontainer.ComponentContainer

Functions
uihtml | uifigure

Related Examples

- “Develop Custom UI Components Programmatically” on page 12-2
- “Create HTML Content in Apps” on page 4-23

Create Custom UI Components in App Designer

Create a Simple Custom UI Component in App Designer

In addition to the UI components that MATLAB provides for building apps, you can create custom UI components to use in your own apps or to share with others. Starting in R2022a, you can interactively create custom UI components in App Designer.

Some benefits of creating custom UI components include:

- Modularization — Separate the display and code of large apps into independent, maintainable pieces.
- Reusability — Provide a convenient interface for adding and customizing similar components in apps.
- Flexibility — Extend the appearance and behavior of existing UI components.

Component Creation Overview

When you design and create a custom UI component, there are two users of your component to consider: app creators and app users. App creators use your component when building an app, whereas app users interact with your component when running an app. Because these two types of users use your component in different ways, there are additional considerations to take into account when designing a custom component as opposed to designing an app.

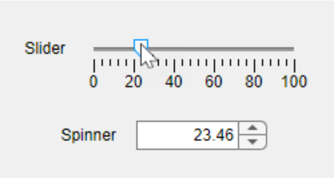
To provide a good experience for app creators who use your component to build an app:

- Provide an interface for users to customize the appearance and behavior of the component in an app by creating public properties.
- Enable users to program a response to interactions with the component by creating public callbacks.
- Ensure that the component is robust to the different ways in which users can incorporate it into their apps, and provide feedback, such as descriptive error messages, when a user attempts to use it in an unintended way.

To provide a good experience for app users who interact with your component in an app:

- Design the component appearance so that users can understand its purpose.
- Program the basic behavior of the component so that it is consistent across all apps that use it.

Learn how to create a custom UI component in App Designer by walking through the process of creating a slider-spinner UI component that provides a flexible interface to change a numeric value. When you have completed the process, you will be able to use the slider-spinner component in an App Designer app in the same way you use existing UI components.



You can either create the slider-spinner component by following the interactive tutorial in the App Designer environment or by following the steps on this page.

To run the interactive component tutorial in App Designer, open the App Designer Start Page. In the **Custom UI Components** section, click **Show examples**, and then select **Interactive Component Tutorial**.

App Designer Component Tutorial Steps

This table provides a quick reference for the App Designer interactive component tutorial.

Step Number	Step Instruction
1	Let's build a simple custom UI component that consists of a slider and a spinner with their values connected together. Drag a **Slider** component onto the canvas.
2	Drag a **Spinner** component onto the canvas.
3	Create a public property named `Value` to allow app creators to set the slider-spinner component value when they use the component to build an app. Select the component node in the **Component Browser** and click the ➕ button to add a new public property.
4	Fill out the dialog with the following values and then click **OK** to create the property: **Name**: `Value` **Data Type**: `double` **Default Value**: `0`
5	App Designer has a design view for designing your component and a code view for programming your component. Click **Code View** to begin programming your component.
6	Create a callback function for the slider to respond when a user changes the slider value. In the **Component Browser**, right-click `comp.Slider` and select **Callbacks > Add ValueChangedFcn callback**.
7	When the slider value changes and this callback function is executed, update the `Value` property of your component to match the new slider value. Replace the callback code with the following: `comp.Value = event.Source.Value;`
8	Next, do the same thing for the spinner. Instead of creating a new callback function, you can reuse the slider callback. In the **Component Browser**, right-click `comp.Spinner` and select **Callbacks > Select existing callback**. Then, select the existing callback for the slider from the dialog.

Step Number	Step Instruction

9	Your component's `Value` property now updates when the value of the slider or the spinner changes. Next, conversely, update the slider and spinner value when someone sets your component's `Value` property programmatically.
	Write the following code to do this in the `update` function, which executes every time the value of a public property changes:
	`comp.Slider.Value = comp.Value;`
	`comp.Spinner.Value = comp.Value;`

10	Create a public callback to allow users of your component to program a response to an interaction in the context of their app. App Designer creates a public callback when you add an event. Click on the **Events** tab and then click the + button to add a new event.
11	When you add an event, App Designer creates a public callback of the same name with `Fcn` appended. This callback is available when the component is added to an app. Specify the name of the event to be `ValueChanged`.
12	To trigger the `ValueChangedFcn` callback at the right time, you need to notify the event whenever you change your component's `Value` property. Add the following code to the end of the `SliderValueChanged` callback: `notify(comp, 'ValueChanged')`
13	Click ▶ **Run** to save and run the component to verify the its interactive behavior.

14	To verify that your component's `ValueChangedFcn` callback is triggered when the component value changes, execute the following code in the MATLAB Command Window:
	`comp = tutorialComponent;`
	`comp.ValueChangedFcn = @(src,event)disp(src.Value);`

15	Congratulations, you just built your first custom UI component! Click 🛠 **Configure** to enable your component to appear in the App Designer Component Library. You can use the component in your apps or share it with others.

Create Custom UI Component

To create a custom UI component in App Designer, first open a new blank component. Open the App Designer Start Page, and in the **Custom UI Components** section, click **Blank Component**. Save the component file as `myComponent.mlapp`. Then, follow these steps to build the slider-spinner component with connected values:

1. "Design Component Appearance" on page 13-5 — Lay out your custom component using existing UI components in App Designer **Design View**.

2. “Design Component Interface” on page 13-5 — Provide options for an app creator to customize the appearance and behavior of your custom component to suit the needs of the app. Do this in two steps:

   a. “Create and Configure Public Properties” on page 13-5 — Enable the app creator to specify aspects of the component appearance and behavior in an app.
   b. “Create and Configure Public Callbacks” on page 13-7 — Enable the app creator to program a response when an app user interacts with the component in an app.
3. “Verify Component Behavior” on page 13-8 — Ensure your custom component looks and behaves as intended.
4. “Configure Component for Use in Apps” on page 13-9 — Specify how your custom component appears in the App Designer **Component Library**.

Design Component Appearance

Design your custom component appearance in **Design View**.

To design the slider-spinner appearance, first drag a **Slider** component from the **Component Library** onto the canvas. Then, drag a **Spinner** component onto the canvas and position it below the slider.

When you lay out your custom component in **Design View**, App Designer generates the code that creates the underlying UI components in the `setup` function in **Code View**. This function is run once when the custom component object is created in an app. If you have additional startup tasks that you would like to execute once at setup, such as plotting data or initializing default values, you can create a `PostSetupFcn` callback for the custom component. For more information, see “Define Custom UI Component Startup Tasks in App Designer” on page 13-11.

Design Component Interface

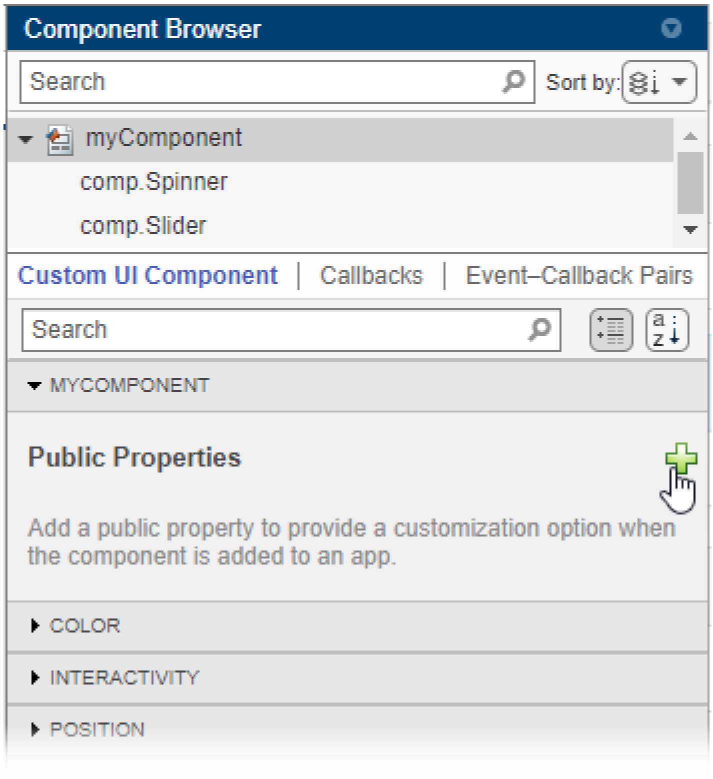
Design your custom component interface so that an app creator can specify how the component appears and behaves within their app. There are two aspects of the component interface to consider:

- Public properties — These are properties of the component that can be set and queried when the component is added to an app. Create public properties to provide component customization options and to expose information about the component to an app creator.
- Public callbacks — These are callbacks of the component that can be accessed when the component is added to an app. Create public callbacks to allow an app creator to program a response to a specific component interaction in the context of their app.

Create and Configure Public Properties

To add a public property to your custom component, first create the property and specify its default value, data type, and associated validation functions. Then, write code to connect the property value and the custom component appearance and behavior. Update the public property value when an app user interacts with the component, and update the underlying components when an app creator programmatically sets the public property value.

For the slider-spinner component, create a public property named `Value` to allow app creators to set the slider-spinner value when they use the component in app. Select the component node in the **Component Browser** and click the ✚ button.

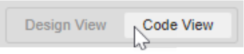

Component Browser
Search
Sort by:
myComponent
comp.Spinner
comp.Slider
Custom UI Component | Callbacks | Event–Callback Pairs
Search
MYCOMPONENT
Public Properties
Add a public property to provide a customization option when the component is added to an app.
COLOR
INTERACTIVITY
POSITION

When you add a new public property, App Designer opens a dialog box that lets you specify property details. Fill out the dialog box with these values:

- **Name** — Enter `Value` as the property name.
- **Data Type** — Enter `double` as the property data type.
- **Default Value** — Enter `0` as the default property value.

Click **OK** to create the property.

Once you create the `Value` public property, write code to link the property to the slider-spinner component appearance. Navigate to **Code View** using the button in the upper-right corner of the canvas.



When you create a custom UI component, App Designer creates a class definition file with `matlab.ui.componentcontainer.ComponentContainer` as the superclass. Use the **Code View** editor to write code to program your component in this file. Access the custom component object in your code by using `comp`.

First, update the `Value` property whenever an app user interacts with the slider or the spinner. Perform this update using these steps:

1. Create a callback for the slider component. Right-click `comp.Slider` in the **Component Browser** and select **Callbacks > Add ValueChangedFcn callback**.
2. Update the `Value` property within the `SliderValueChanged` callback function. Replace the code in the callback function with this code:

```
comp.Value = event.Source.Value;
```

3. Assign the same callback function to the spinner component. Right-click `comp.Spinner` in the **Component Browser** and select **Callbacks > Select existing callback**. Use the dialog box to assign the `SliderValueChanged` function to the spinner `ValueChangedFcn` callback.

At this point, your code updates the slider-spinner `Value` property whenever the underlying component values change. Next, conversely, write code to update the underlying spinner and slider components whenever an app creator sets the slider-spinner `Value` property. Perform this update inside the `update` function of your component code. The `update` function executes whenever the value of a public property of the custom component changes, so this ensures that the underlying spinner and slider values always match the value of the custom component.

Add this code to the `update` function:

```
comp.Slider.Value = comp.Value;
comp.Spinner.Value = comp.Value;
```

For more information about creating and configuring public properties, see “Create Public Properties for Custom UI Components in App Designer” on page 13-13.

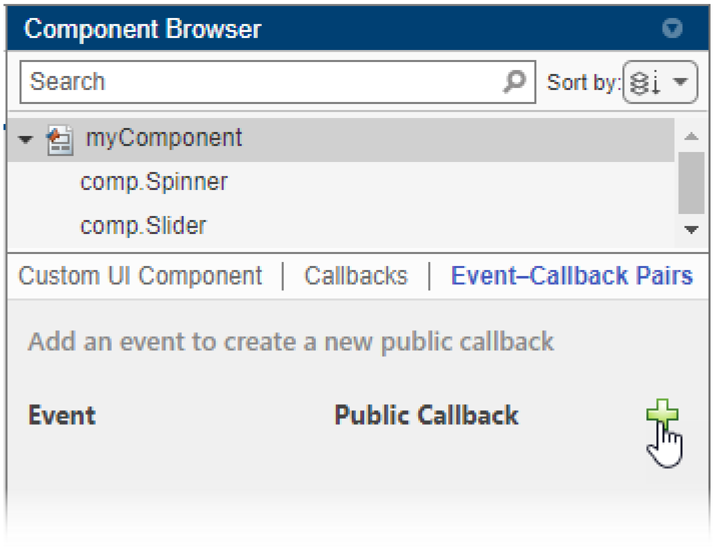
Create and Configure Public Callbacks

Create public callbacks for your custom component to enable app creators to write code in their apps to respond to specific interactions. To create a public callback, add an event. An event is a notice that is broadcast when an action occurs. When you add an event for your custom component, App Designer creates a public callback associated with the event. Then, write code to trigger the event, which executes the associated public callback.

In the slider-spinner component, create an event named `ValueChanged` and write code to trigger the event and execute the associated `ValueChangedFcn` callback whenever an app user changes the value of the slider or the spinner in a running app. This lets app creators to use the `ValueChangedFcn` callback in their app. For instance, an app creator might write a callback function to plot some data whenever the app user changes the slider-spinner value.

Use these steps to add and trigger the `ValueChanged` event:

1. Select the **Event–Callback Pairs** tab in the **Component Browser** and click the ✚ button.



Component Browser

Search Sort by:

myComponent
comp.Spinner
comp.Slider

Custom UI Component | Callbacks | Event–Callback Pairs

Add an event to create a new public callback

Event Public Callback

2 In the Add Event–Public Callback Pair dialog box, enter the event name as `ValueChanged`, and click **OK**. App Designer creates a public callback of the same name with the letters `Fcn` appended. So in this case, the public callback is named `ValueChangedFcn`.

3 Write code to ensure that the callback is executed at the appropriate moment. Do this by calling the `notify` function on the component object and specifying the event name. Here, the callback should be executed when an app user interacts with the slider or the spinner. In the `SliderValueChanged` function, add this code:

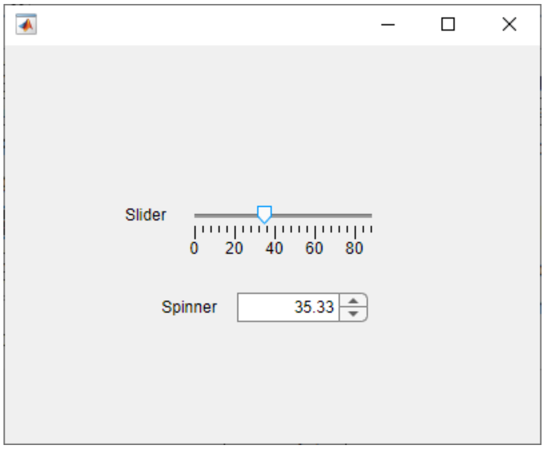
```
notify(comp,"ValueChanged")
```

For more information about creating and executing public callbacks, see “Create Callbacks for Custom UI Components in App Designer” on page 13-21.

Verify Component Behavior

To see what your component looks like in a running app, save the component and then click ▷ **Run**. App Designer displays a UI figure window that contains your custom component.

Run the slider-spinner component, and interact with it to verify that it looks and behaves as expected.



Spinner 35.33

Next, verify that the `ValueChangedFcn` callback is executed when the component value changes. Create an instance of your component programmatically by specifying the component file name at the MATLAB Command Window and returning the component object as a variable. Enter this code in the Command Window to create a slider-spinner component and assign a callback function:

```
comp = myComponent;
comp.ValueChangedFcn = @(src,event)disp(src.Value);
```

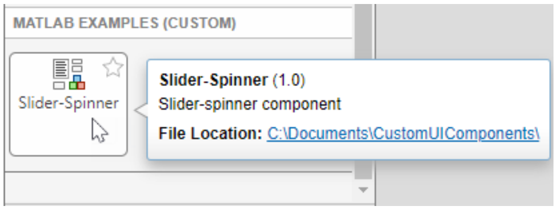
Interact with the slider and spinner. The value of the custom component is displayed in the Command Window.

Configure Component for Use in Apps

To use your custom UI component in an App Designer app or to share it for others to use, follow these steps:

1. In the **Designer** tab, click **Configure for Apps**.
2. Fill out the App Designer Custom UI Component Metadata dialog box, and then click **OK**.
3. In the confirmation dialog box, click **Add to Path** to add the component and generated `resources` folder to the MATLAB path.
4. In the **Designer** tab, click **New** and select **Blank App**.

The component appears in the **Component Library** of the app, under the category specified in the dialog box.

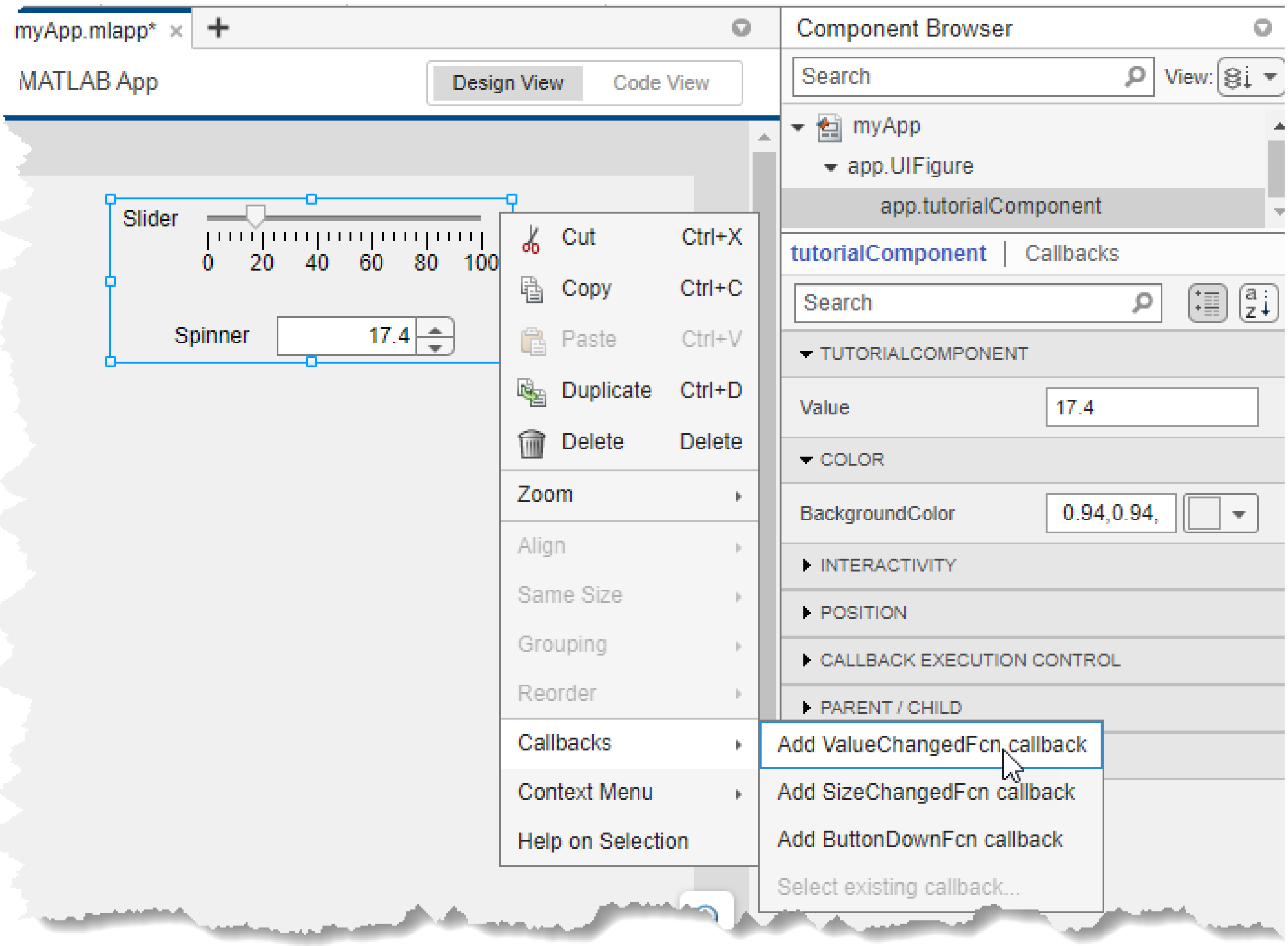


MATLAB EXAMPLES (CUSTOM)

Slider-Spinner

Slider-Spinner (1.0)
Slider-spinner component
File Location: C:\Documents\CustomUIComponents\

Drag a slider-spinner component onto the canvas. You can set public properties of the component using the **Component Browser**, and you can assign public callbacks to program the component behavior in the app.



Spinner 17.4

Value 17.4

BackgroundColor 0.94,0.94,

For more information about configuring and sharing your custom UI component, see “Configure Custom UI Components for App Designer” on page 12-17.

See Also

Classes
`matlab.ui.componentcontainer.ComponentContainer`

Related Examples

- “Create Public Properties for Custom UI Components in App Designer” on page 13-13
- “Create Callbacks for Custom UI Components in App Designer” on page 13-21
- “Configure Custom UI Components for App Designer” on page 12-17

Define Custom UI Component Startup Tasks in App Designer

When you create a custom UI component in App Designer, you and other app creators can use the component in an app. You can write code to perform one-time startup tasks that executes when an app containing your component first starts up, but before an app user interacts with the app. Perform these startup tasks in a `PostSetupFcn` callback of the custom component object. For example, you can use a `PostSetupFcn` callback to initialize a public property for your component or to display data in a plot or a table.

Component Setup Overview

When you lay out a custom UI component in **Design View**, App Designer generates code to create and configure the initial custom component layout and behavior. The code creates the underlying UI components that make up your custom component and sets properties and callbacks of those underlying components. App Designer adds this startup code to the `setup` function of the custom component class file in **Code View**. The `setup` function executes once when an instance of the custom component is created, such as when an app containing the custom component is run.

If you add a `PostSetupFcn` callback to your custom UI component, the code in the function is executed as part of the component `setup` function, after the setup code generated by App Designer.

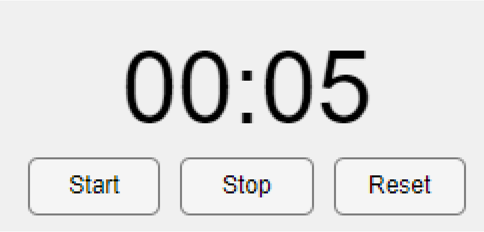
Add PostSetupFcn Callback

To write code to perform startup tasks when your custom component is created, add a `PostSetupFcn` callback to the component. There are several ways to create a `PostSetupFcn` callback.

- Right-click the component node at the top of the **Component Browser** hierarchy, and select **Callbacks** > **Add PostSetupFcn callback**.
- Select the component node at the top of the **Component Browser**, and then select the **Callbacks** tab. Under **Private Callbacks**, expand the drop-down list next to **PostSetupFcn** and select **<add PostSetupFcn callback>**.
- In **Code View**, in the **Editor** tab of the toolstrip, click **Callback**.
- In **Code View**, in the **Callbacks** tab of the **Code Browser**, click the ✚ button.

Example: Timer Component That Performs Startup Tasks

This example shows how to program the startup tasks and behavior of a custom timer component. The `PostSetupFcn` creates and configures the underlying `timer` object that controls the component behavior.



The component consists of:

- Text that counts down the time remaining on the timer
- `Start`, `Stop`, and `Reset` buttons for app users to interact with the timer
- Public properties named `Seconds` and `Minutes` for app creators to specify the length of the timer
- A public callback named `TimerEndedFcn` for app creators to program a response to the timer reaching zero

To explore the component code, open the `TimerComponent.mlapp` file in App Designer.

See Also

Classes
`matlab.ui.componentcontainer.ComponentContainer`

Related Examples

- “Create a Simple Custom UI Component in App Designer” on page 13-2
- “Create Public Properties for Custom UI Components in App Designer” on page 13-13
- “Create Callbacks for Custom UI Components in App Designer” on page 13-21

Create Public Properties for Custom UI Components in App Designer

When you develop a custom UI component in App Designer, create properties to store data and to allow users to modify the appearance and behavior of the component when using it in an app. Properties of a custom UI component can be either public or private. Public and private properties serve different purposes, and the type you create depends on your goal.

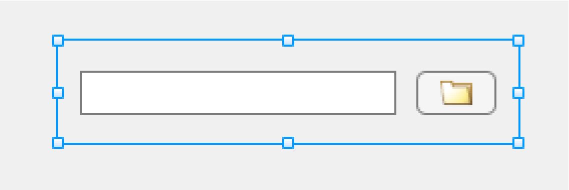
- *Public properties* are accessible when the component is used in an app. Create a public property to provide customization options that app creators can set when building an app with your component.
- *Private properties* are accessible only within your component code. Create a private property to store data and to share that data between callbacks in your component.

Because public properties can be set and queried by others when they create an app that uses your component, it is a best practice to validate the values that a public property of your component can accept.

To add a public property to a component, use these steps:

1. “Create New Public Property” on page 13-13 — Create a robust and user-friendly public property by initializing and validating the property values.
2. “Configure Public Property” on page 13-16 — Write code to link the public property value to the custom component appearance and behavior.
3. “Verify Public Property” on page 13-17 — Verify that setting the property programmatically updates the custom component and that the property appears when the component is used in an App Designer app.

The example on this page shows how to create, configure, and verify a public property for a custom UI component that allows users to select a file from their system. The public property `Path` stores the path to the file that an app user selects.



To explore the finished `FileSelector` component in App Designer, see “Full Example: FileSelector Component” on page 13-19.

Create New Public Property

While building a custom UI component, you can create a new public property in multiple ways, depending on where in App Designer you are working:

- In the **Component Browser**, select the top-level component node. Under the **Custom UI Component** tab in the **Property Inspector**, click the ➕ button.

- In the **Editor** tab in **Code View**, click **Property** and select **Public Property**.
- In the **Code Browser** in **Code View**, under the **Properties** tab, click the ✚ button.

When you add a new public property, App Designer opens a dialog box that lets you specify property details. This table lists the dialog box options.

Option	Details
Name	The name of the public property. App creators can use the property name to set and access the value of the property in an app.
Description	A description of what the property does. The description is inserted as a comment next to the property definition in **Code View**.
Data Type	The data type of the value that the property stores. Select a common data type from the drop-down list in the dialog box, or type any MATLAB data type. For the public property to appear in the **Property Inspector** when an app creator uses your component in App Designer, you must specify a data type or default value. Only certain data types allow for in-place property editing in **Design View**. For a list of these supported data types, see “Configure Custom UI Components for App Designer” on page 12-17. You can specify a data type that is not included in the list, but the app creator will have to set the property value programmatically in **Code View** instead of interactively in the inspector.
Default Value	The default value of the public property when the component is added to an app. For the public property to appear in the **Property Inspector** when an app creator uses your component in App Designer, you must specify a data type or default value.
Size	The size of the data that the property stores. Specify the size of the data as the number of rows by the number of columns.

Option	Details
Validation Functions	The validation functions applied to the property value when the app creator sets it. Use validation functions to help app creators avoid unexpected behavior when using your component and to provide descriptive error messages when a user specifies an invalid property value. Some common types of validation include: • Validating that a numerical property value is within a specific range • Validating that a property value is one of a finite set of options • Validating that a property value is not missing or empty For a list of the available validation functions and their uses, see “Property Validation Functions”.
MATLAB Property Attributes	The attributes associated with the property. Property attributes control characteristics like access, data storage, and visibility of the property. The `Access` property attribute is required and is specified as `public` by default when you add a new public property. For a list of property attributes and their uses, see “Property Attributes”

To edit an existing public property that you have created, select the top-level component node in the **Component Browser**, click the ⋮ button next to the property name, and select **Edit**. This selection brings up the Edit Property dialog box with the existing property configurations. You can update any configurations and click **OK** to apply the changes.

To create the `Path` property for the `FileSelector` component, navigate to the **Component Browser** and select the `FileSelector` node. Then, click the ✚ button.

Fill out these fields in the Add Property dialog box:

- **Name** — Enter `Path`.
- **Data Type** — Select `string` from the drop-down list.
- **Validation Functions** — Select the `mustBeFile` validation function from the drop-down list to validate that the value represents a valid file.
- **MATLAB Property Attributes** — Select the `AbortSet` attribute. This selection specifies that MATLAB does not set the property value if the new value is the same as the old one. Specifying `AbortSet` can improve the performance of your component. Because it is possible that an app creator will program the `FileSelector` component to perform an expensive operation, such as processing file data, after an app users selects a new file, this attribute ensures that any code in an app runs only if the app user selects a new file.

Do not specify a default value for the `Path` property because the property value depends on the files on an app user's system.

Configure Public Property

After you create a public property, write code to associate the property value with the appearance and behavior of the custom component. Often, you can do this in two steps:

1. "Update Public Property Value" on page 13-16 — Update the value of the public property when an app user interacts with the custom component.
2. "Update Underlying Components" on page 13-16 — Update the display or behavior of the underlying components when an app creator sets the value of the public property.

Update Public Property Value

To update the value of a public property when an app user interacts with the component, create a callback associated with an underlying UI component, and update the public property value in the callback function in **Code View**. You can reference the public property in your code by using the pattern `comp.`*`PublicPropertyName`*.

In the `FileSelector` component, update the value of the `Path` property in response to two interactions:

- When the user specifies a new file by typing a path to the file in the edit field
- When the user selects a new file interactively by pressing the button

For the edit field interaction, create a `ValueChangedFcn` callback for the edit field in the `FileSelector` component:

1. Right-click the `comp.EditField` node in the **Component Browser** and select **Callbacks** > **Add ValueChangedFcn callback**.
2. Add code to the `EditFieldValueChanged` function in **Code View** to update the value of the `Path` property to align with the path that is displayed in the edit field:

   ```
   comp.Path = comp.EditField.Value;
   ```

For the button interaction, create a `ButtonPushedFcn` callback for the button in the `FileSelector` component:

1. Right-click the `comp.Button` node in the **Component Browser** and select **Callbacks** > **Add ButtonPushedFcn callback**.
2. Add code to the `ButtonPushed` function in **Code View** to open a file selection dialog box and then to update the value of the `Path` property based on the file that the user selects:

   ```
   [fileName,pathName] = uigetfile("","Select a file");
   if ~isequal(fileName,0)
       comp.Path = fullfile(pathName,fileName);
   end
   ```

Update Underlying Components

To update the display or behavior of the underlying components when an app creator sets the value of a property, write code in the `update` function in **Code View**. This function executes whenever the value of a property of the custom component changes.

In the `update` function of the `FileSelector` component, write code to display the value of the `Path` property in the edit field:

```
function update(comp)
    comp.EditField.Value = comp.Path;
end
```

Verify Public Property

You can verify that a public property appears and behaves as expected in multiple ways:

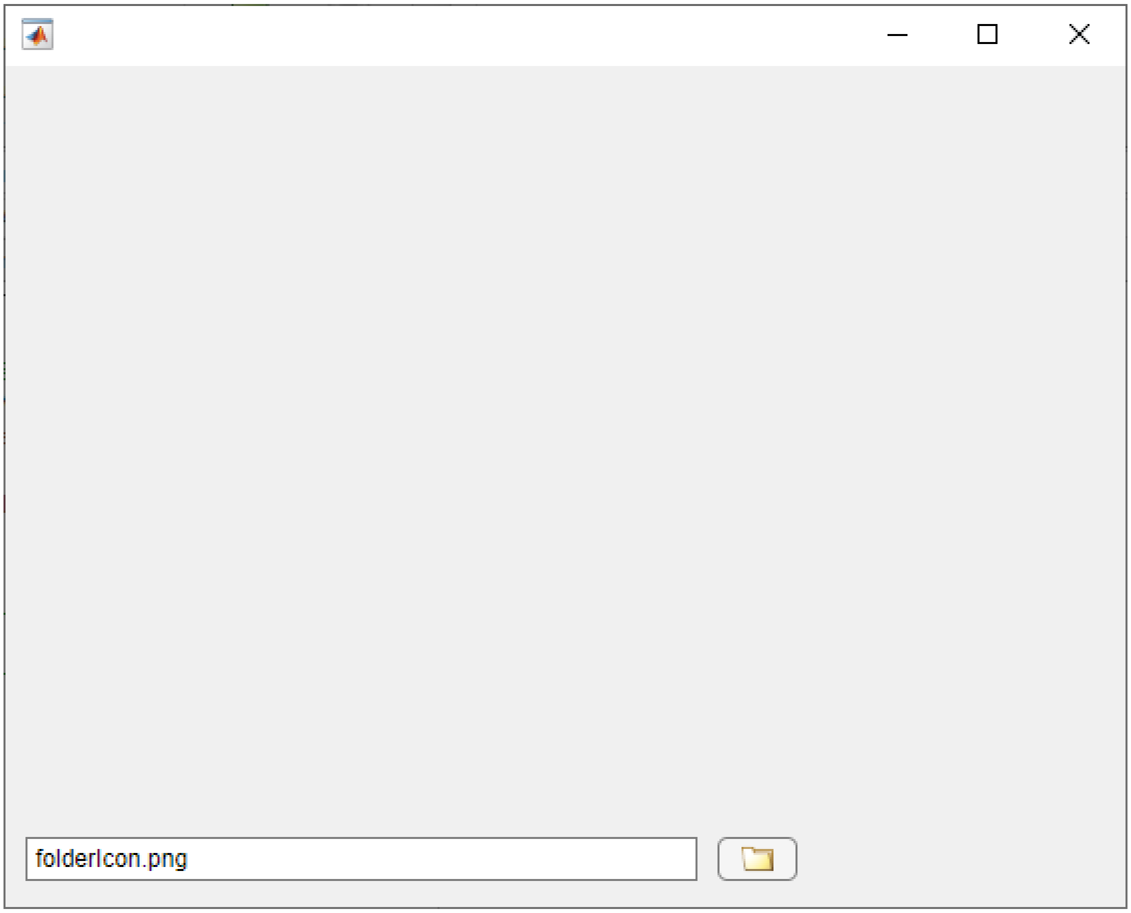
- Create an instance of the custom component from the MATLAB Command Window and set the property programmatically.
- Add the custom component to an App Designer app and set the property using the Property Inspector.

Verify Public Property Programmatically

Create an instance of the custom component by entering the name of the file in the MATLAB Command Window. Return the component object as a variable and use the variable to set the property programmatically.

Verify that the `Path` property is linked to the `FileSelector` component display by navigating to the folder where the `FileSelector.mlapp` file is saved and entering this code into the MATLAB Command Window:

```
comp = FileSelector;
comp.Path = "folderIcon.png";
```


folderIcon.png

The path is displayed in the edit field.

You can also verify that your property validation is working as intended. For example, if you set the `Path` property to a nonexistent file, MATLAB detects the invalid file using the `mustBeFile` validation function and throws a helpful error.

```
comp.Path = "myFile.txt";
```

```
Error setting property 'Path' of class 'FileSelector'. The following files do not exist: 'myFile.txt'.
```

Verify Public Property in App

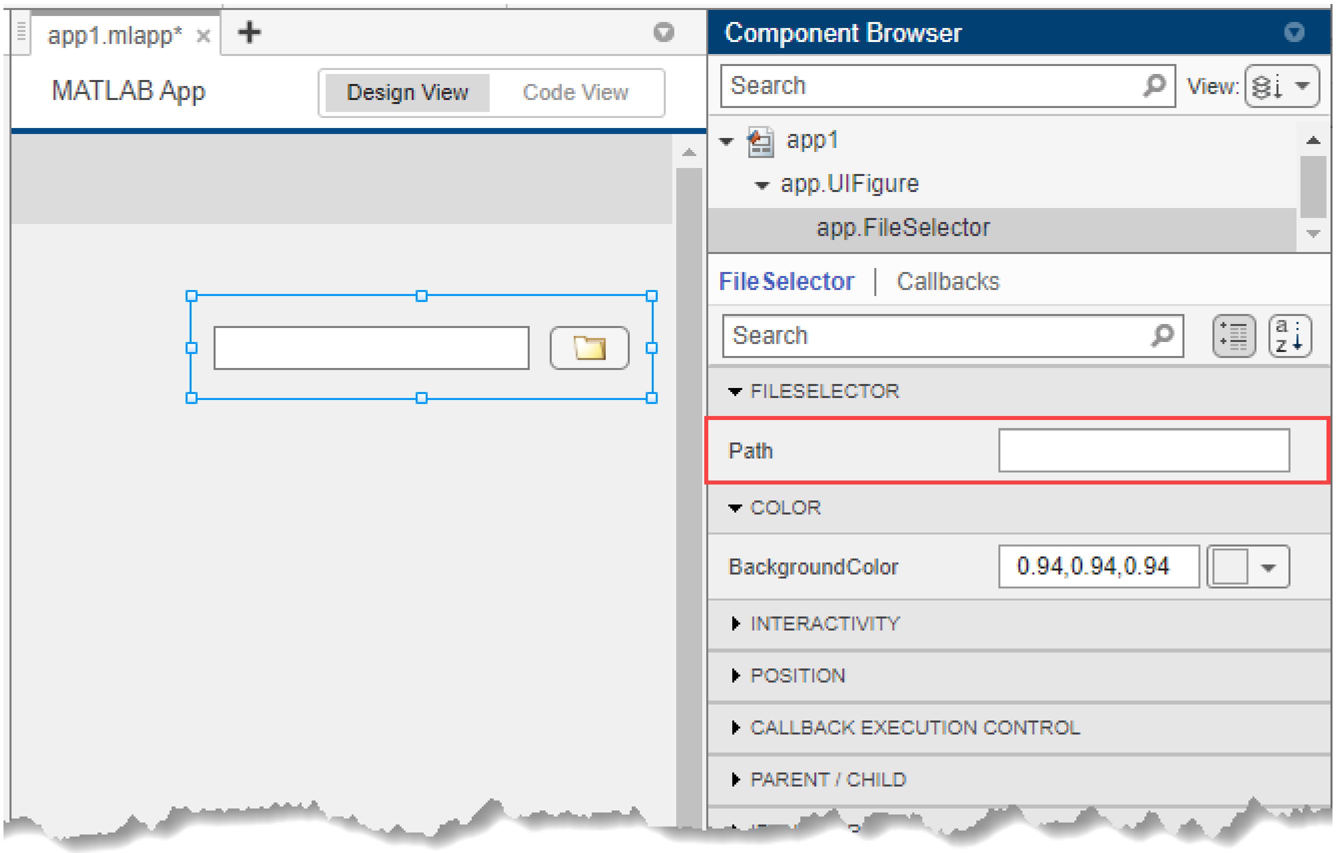
You can also verify that a public property appears and behaves as expected by configuring the custom component for use in App Designer apps. You can then add the component to an app and interact with it using the **Component Browser**.

With the `FileSelector.mlapp` file open in App Designer, follow these steps to add the component to an App Designer app.

1. In the **Designer** tab, click **Configure for Apps**.
2. Fill out the App Designer Custom UI Component Metadata dialog box, and then click **OK**.
3. In the confirmation dialog box, click **Add to Path** to add the component and generated `resources` folder to the MATLAB path.

4 In the **Designer** tab, click ✚ **New** and select **Blank App**.

5 Drag the file selector component from the **Component Library** onto the app canvas. When the component is selected in the **Component Browser**, the `Path` property appears in the Property Inspector.



Note Avoid using your component in an App Designer app while you are actively developing your component. If you make a change to your component code while the component is being used in an open App Designer app, you must restart App Designer to see the changes reflected within the app. For more information, see "Verify Behavior of Custom UI Components in App Designer" on page 13-36.

Full Example: FileSelector Component

This example shows the full `FileSelector` custom UI component code created in App Designer. The component is made up of a button that app users can press to select a file on their system and an edit field that displays the path to the selected file. The component interface consists of a public property and a public callback:

- `Path` — A public property that stores the path to the selected file

- `PathChangedFcn` — A public callback that executes whenever a user selects or enters a new file

See Also

Classes
`matlab.ui.componentcontainer.ComponentContainer`

Related Examples

- “Create a Simple Custom UI Component in App Designer” on page 13-2
- “Create Callbacks for Custom UI Components in App Designer” on page 13-21
- “Write Property Set Methods for Custom UI Components in App Designer” on page 13-26
- “Configure Property Display for Custom UI Components in App Designer” on page 13-50

Create Callbacks for Custom UI Components in App Designer

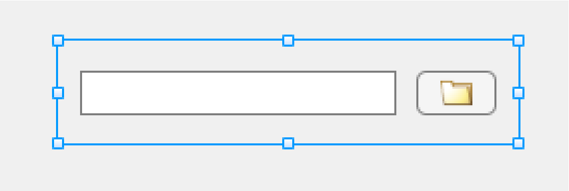
When you develop a custom UI component in App Designer, you can use callbacks to provide options for customizing the component behavior in an app, or to program your own response to user interactions in the component code.

To enable an app creator to program a response to an interaction with your custom component in their app, create a public callback. A public callback is a callback for your component that is accessible when the component is used in an app. For example, if you create a `ValueChangedFcn` public callback for an IP address component, app creators can use this callback to write code to update their app whenever an app user enters a new IP address.

To program the behavior of your component that does not change, regardless of how the component is used, create an underlying component callback in your custom component code. These callbacks are not accessible by an app creator who uses your component in their app. For example, if your custom component contains a button that always opens a dialog box, create a `ButtonPushedFcn` callback for that underlying button component and write code to open the dialog box. App creators cannot access or change this callback functionality in their own apps.

The example on this page shows how to create and configure a public callback for a custom UI component that allows users to select a file from their system. The `FileSelector` component consists of these elements:

- A button that opens a file selection dialog box
- An edit field that displays the path to the selected file
- A public property named `Path` that stores the selected file path



Create a public callback named `PathChangedFcn` for the `FileSelector` component using these steps:

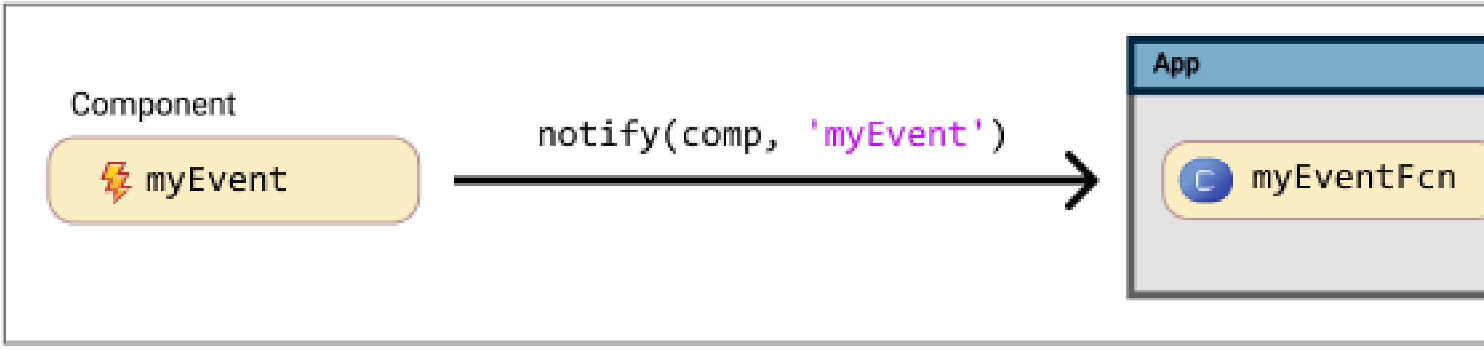
1. "Create New Event" on page 13-22 — Add an event to create a public callback.
2. "Notify Event to Execute Callback" on page 13-23 — Write code to execute the `PathChangedFcn` callback when an app user interactively selects a new file.
3. "Verify Callback" on page 13-24 — Verify that the callback appears and behaves as expected by programmatically assigning a callback function in the MATLAB Command Window and by adding the component to an App Designer app.

To explore and use the finished `FileSelector` component, see "Full Example: FileSelector Component" on page 13-25.

Relationship Between Events and Public Callbacks

To add a new public callback for your component, you must create an event. An event is a notice that is broadcast when an action occurs. When you create an event for your custom component, App

Designer automatically creates a public callback associated with the event whose name is the event name followed by the letters `Fcn`. You can trigger the event by calling the `notify` function in response to a user interaction, which then prompts MATLAB to execute the callback function that the app creator assigned to the associated public callback.



Create New Event

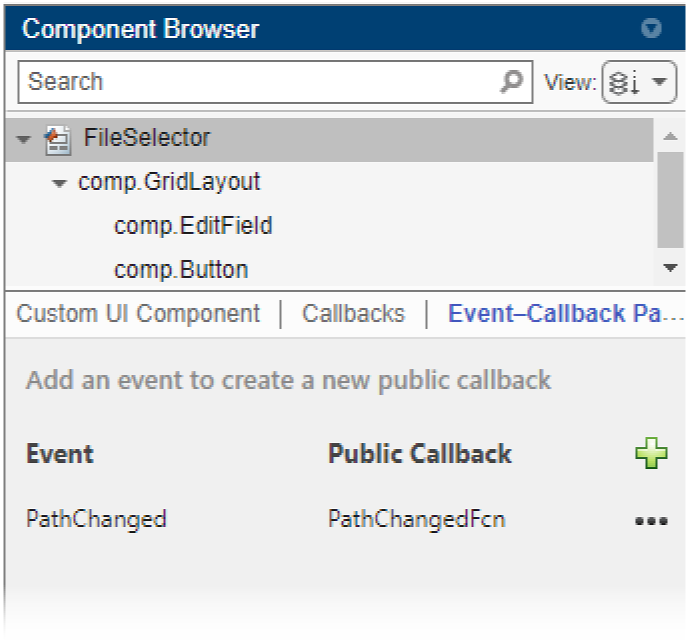
To create a `PathChangedFcn` public callback for the `FileSelector` component, first create a new event. There are multiple ways to create a new event. Choose the option that best suits your workflow based on where in App Designer you are currently working:

- In **Design View**, in the **Component Browser**, select the top-level component node. Under the **Events-Callback Pairs** tab, click the ✚ button.
- In **Code View**, in the **Editor** tab of the toolstrip, click **Event**.
- In **Code View**, in the **Events** tab of the **Code Browser**, click the ✚ button.

Fill out the resulting dialog box with the name of the event and an optional event description:

- Enter `PathChanged` as the event name. App Designer creates an associated public callback named `PathChangedFcn`.
- Specify in the event description that the event is triggered when a user selects a new file. App Designer adds this text as a comment in the `event` block in **Code View**.

You can view the event–public callback pairs for your component by navigating to the **Event-Callback Pairs** tab in the **Component Browser**.



Add an event to create a new public callback

Event	Public Callback
PathChanged	PathChangedFcn

Notify Event to Execute Callback

After you create an event, write code to ensure that the associated public callback is executed at the appropriate moment. To execute a callback, trigger the associated event in your custom component code by calling the `notify` function. Use this syntax:

```
notify(comp,"EventName")
```

For any event, you can copy its associated `notify` code from one of multiple locations within App Designer:

- In the **Code Browser**, in the **Events** tab, right-click the event name and select **Copy Code to Trigger Event**.
- In the **Component Browser**, in the **Event-Callback Pairs** tab of the main component node, click the ••• button next to the event name and select **Copy Code to Trigger Event**.

In general, a public callback should be executed when an app user performs a specific interaction, such as pushing a button or typing in an edit field. Therefore, you will often add the `notify` code to an underlying component callback function.

In the code for the `FileSelector` component, execute the `PathChangedFcn` callback whenever an app user clicks the button or edits the file path to select a new file. To do this, trigger the `PathChanged` event in two locations:

- In the `ButtonPushedFcn` callback for the button
- In the `ValueChangedFcn` callback for the edit field

Add this code to the end of each of the callback functions:

```
notify(comp,"PathChanged")
```

If an app that uses your component needs access to additional information about a user interaction when the interaction occurs, you can optionally define custom event data and specify the data when you trigger the event. For more information, see "Define Custom Event Data".

Verify Callback

You can verify that a public callback behaves as expected in multiple ways:

- Create an instance of the custom component from the MATLAB Command Window and assign a callback function programmatically.
- Add the custom component to an App Designer app and assign a callback function interactively.

Verify Callback Programmatically

Save the `FileSelector` component as `FileSelector.mlapp` in your current folder, and then create a `FileSelector` object by entering this command into the Command Window:

```
comp = FileSelector;
```

Assign a `PathChangedFcn` callback to the component.

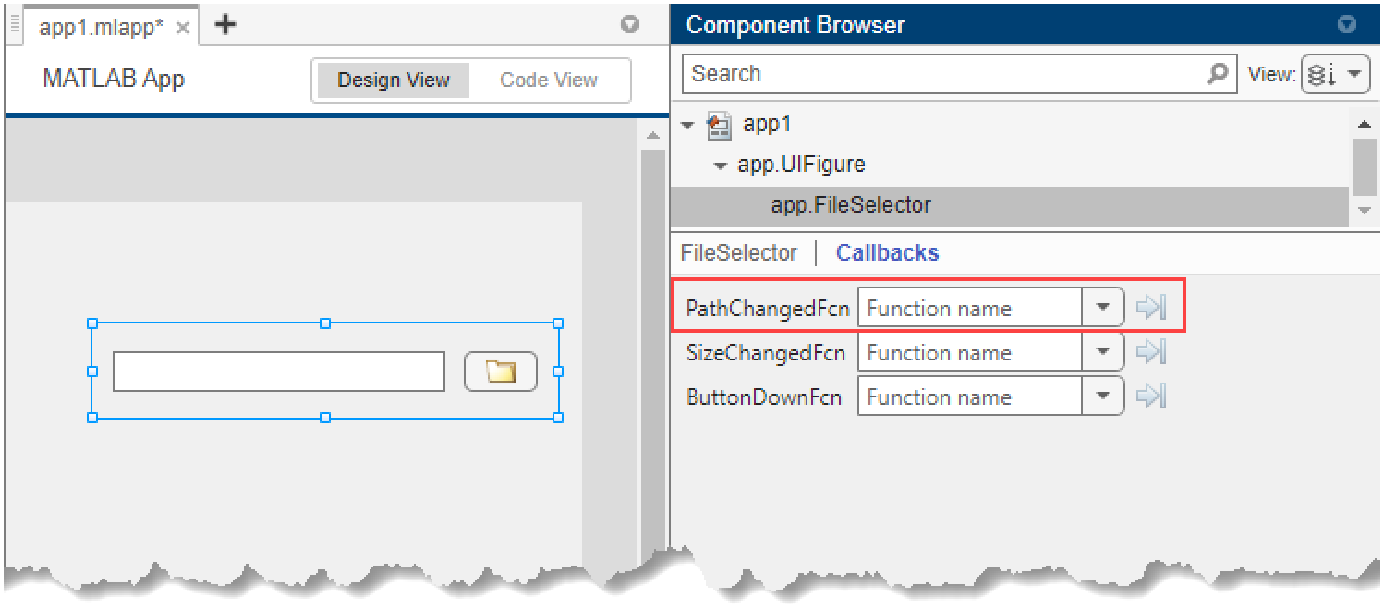
```
comp.PathChangedFcn = @(src,event)disp("File changed");
```

Click the button and select a file using the dialog box. The text "File Changed" displays in the Command Window.

Verify Callback in App

With the `FileSelector.mlapp` file open in App Designer, follow these steps to add the component to an App Designer app and access the public callback:

1. In the **Designer** tab, click **Configure for Apps**.
2. Fill out the App Designer Custom UI Component Metadata dialog box, and then click **OK**.
3. In the confirmation dialog box, click **Add to Path** to add the component and generated `resources` folder to the MATLAB path.
4. In the **Designer** tab, click **New** and select **Blank App**.
5. Drag the file selector component from the **Component Library** onto the app canvas. When the component is selected in the **Component Browser**, the `PathChangedFcn` callback appears in the **Callbacks** tab of the Property Inspector.



PathChangedFcn: Function name
SizeChangedFcn: Function name
ButtonDownFcn: Function name

Note Avoid using your component in an App Designer app while you are actively developing your component. If you make a change to your component code while the component is being used in an open App Designer app, you must restart App Designer to see the changes reflected within the app. For more information, see “Verify Behavior of Custom UI Components in App Designer” on page 13-36.

Full Example: FileSelector Component

This example shows the full `FileSelector` custom UI component code created in App Designer. The component is made up of a button that app users can press to select a file on their system and an edit field that displays the path to the selected file. The component interface consists of a public property and a public callback:

- `Path` — A public property that stores the path to the selected file
- `PathChangedFcn` — A public callback that executes whenever a user selects or enters a new file

See Also

Classes
`matlab.ui.componentcontainer.ComponentContainer`

Related Examples

- “Create a Simple Custom UI Component in App Designer” on page 13-2
- “Define Custom Event Data”
- “Configure Custom UI Components for App Designer” on page 12-17

Write Property Set Methods for Custom UI Components in App Designer

When you create a public property for a custom UI component, one step involves writing code to update the underlying components and graphics objects within your custom component whenever the value of the public property changes. In general, perform this step by writing code in the component `update` function. MATLAB calls the `update` function only when necessary, which can result in performance improvements over writing custom code to perform a similar purpose. For more information about updating properties using the `update` function, see “Create Public Properties for Custom UI Components in App Designer” on page 13-13.

However, you might want to perform certain tasks when one specific property is updated but not when other properties are updated. Because the `update` function executes whenever the value of *any* public property changes, you can instead write code to perform these tasks when a *specific* property changes by defining a set method for that property.

Consider writing a property set method when you want to:

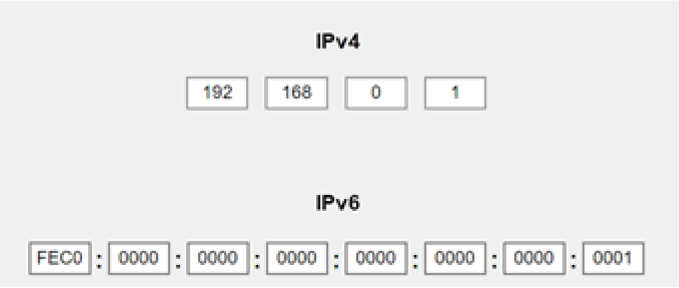
- Perform custom property validation.
- Throw a custom error when the property is set incorrectly.
- Process the property value before storing it.

This example shows how to validate a public property of a custom IP address UI component by writing a set method.

IP Address Component Overview

This example IP address component accepts input formatted using either the IPv4 or IPv6 protocol. The protocol determines how the component is displayed:

- IPv4 — The component contains four numeric edit fields, each with a value between 0 and 255.
- IPv6 — The component contains eight text edit fields, each with four characters representing hexadecimal digits.



The IP address component interface consists of:

- A public property named `Address` to store the value of the IP address
- A public property named `Protocol` to specify the IP address protocol
- A public callback named `AddressChangedFcn` that executes when an app user changes the IP address by typing in an edit field

Because the `Address` property can store either a four-element numeric vector (when the protocol is IPv4) or an eight-element cell array (when the protocol is IPv6), use custom validation logic in a property set method to check whether the `Address` value is valid.

To view the full `IPAddress` component code in App Designer, enter this command in the MATLAB® Command Window:

```
openExample('matlab/IPAddressCustomComponentExample');
```

Create a Property Set Method

To create a new property set method for the `Address` property of the IP address component, use these steps:

1. Create a new public function. In **Code View**, in the **Editor** tab, select **Function** > **Public Function**.
2. In the `methods` block that contains the new function, delete the text `(Access = public)`. Property set methods must be added in a `methods` block with no attributes. This deletion does not change the functionality of the `methods` block because the default value of the `Access` attribute is `public`. For more information, see “Property Get and Set Methods”.
3. Replace the function definition that App Designer creates with this code:

   ```
   function set.Address(comp,val)
   % Write property validation code here
   end
   ```

   The `set.Address` function executes whenever an app creator sets the value of the `Address` public property.

For more information about

Perform Custom Property Validation

Write code in the `set.Address` function to verify that the new property value follows the expected format:

- If the protocol is IPv4, check that the app creator set the property to a vector of four integers between 0 and 255.
- If the protocol is IPv6, check that the app creator set the property to a cell array of eight character vectors, where each character vector represents four hexadecimal digits.

In each case, if the new value is not in the expected format, throw a helpful error to inform the app creator what value the property expects. Finally, set the `Address` property of the component to the new property value.

Add this code to the body of the `set.Address` function:

```
switch comp.Protocol
    % Validate IPv4 address
    case "IPv4"
        if length(val) ~= 4
            error("IPv4 address must have four fields.")
        end
        mustBeInRange(val,0,255)
```

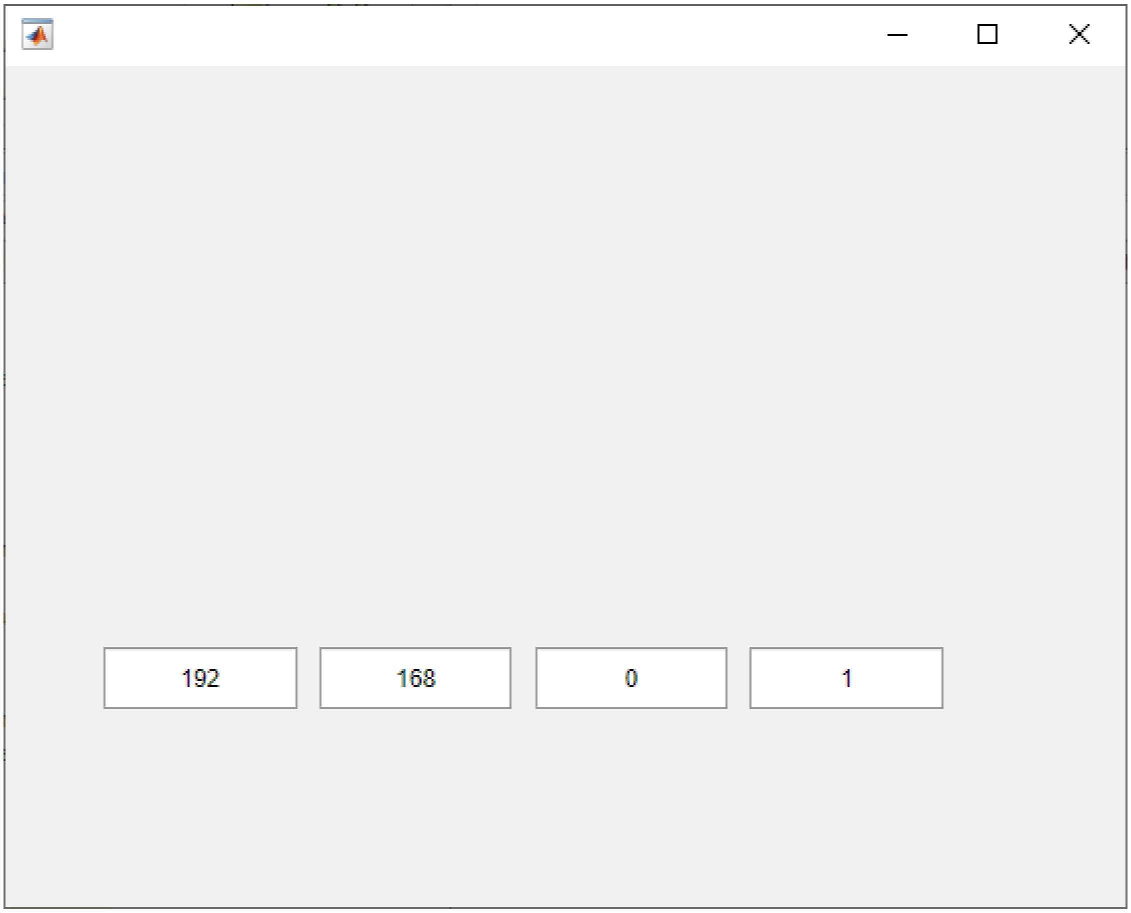
```
    % Validate IPv6 address
    case "IPv6"
        if length(val) ~= 8
            error("IPv6 address must have eight fields.")
        end

        if ~isequal(cellfun('length',val),repmat(4,1,8))
            error("Specify IPv6 field as a four-digit hexadecimal number.")
        end

        try
            hex2dec(val);
        catch
            error("Specify IPv6 field as a four-digit hexadecimal number.")
        end
end
comp.Address = val;
```

Verify Property Validation Behavior

After you have finished developing the IPAddress component, verify the property validation behavior by creating a component object and setting the Address property from the MATLAB Command Window.

Navigate to the folder where the IPAddress.mlapp file is saved. Create an IP address component, specify its position, and return the component object as comp. By default, the component is created using the IPv4 protocol.

```
comp = IPAddress(Position=[50 100 420 31]);
```



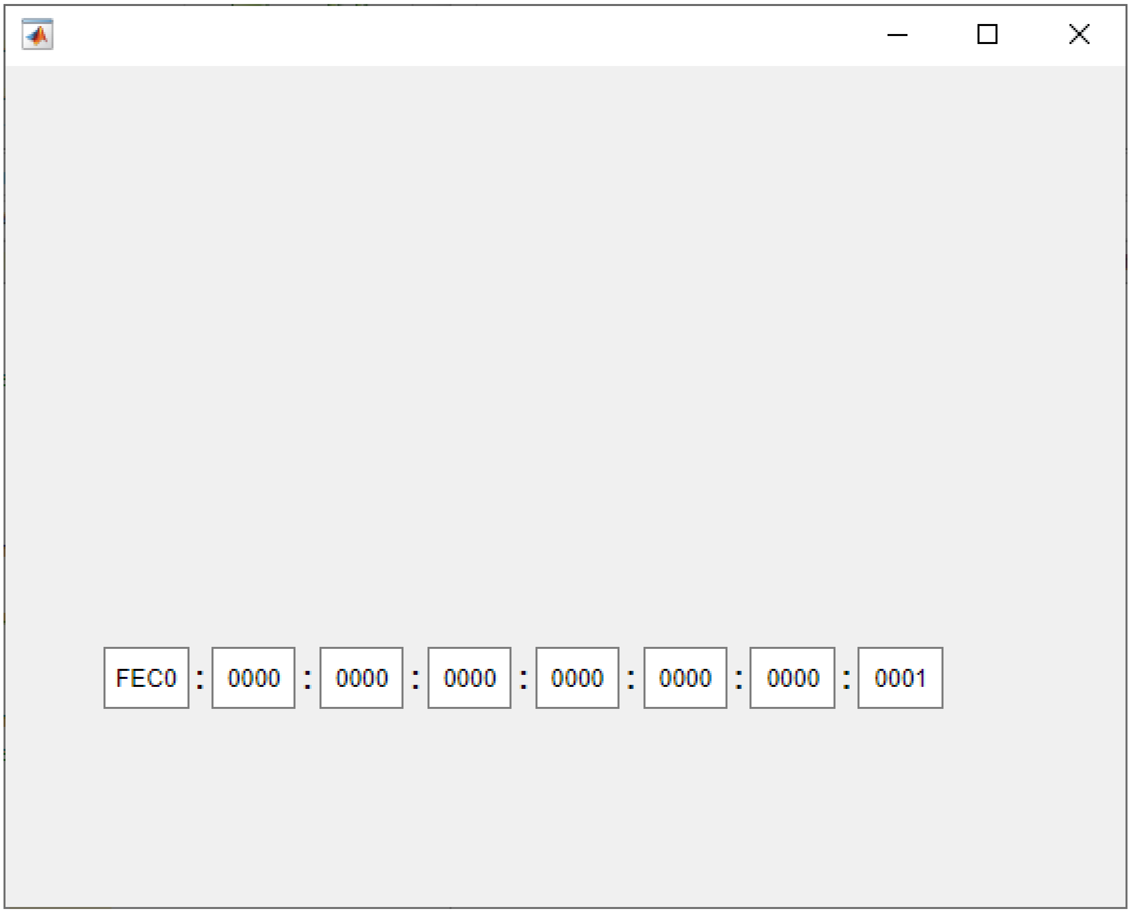
Try to set the `Address` property to a scalar value. An error displays.

```
comp.Address = 10;
```

```
Error using IPAddress/set.Address
IPv4 address must have four fields.
```

Change the component to an IPv6 address component.

```
comp.Protocol = "IPv6";
```



Try to set one field of the IP address to a value that does not represent a four-digit hexadecimal number.

```
comp.Address{1} = '123h';
```

```
Error using IPAddress/set.Address
Specify IPv6 field as a four-digit hexadecimal number.
```

See Also

Related Examples

- "Create Public Properties for Custom UI Components in App Designer" on page 13-13
- "Verify Behavior of Custom UI Components in App Designer" on page 13-36
- "Property Get and Set Methods"

Modularize Your App by Creating a Custom UI Component

As the size and complexity of an app increases, modularizing the app can help organize and manage the code. One modularization method is to separate out self-contained portions of your app layout with common functionality as custom UI components.

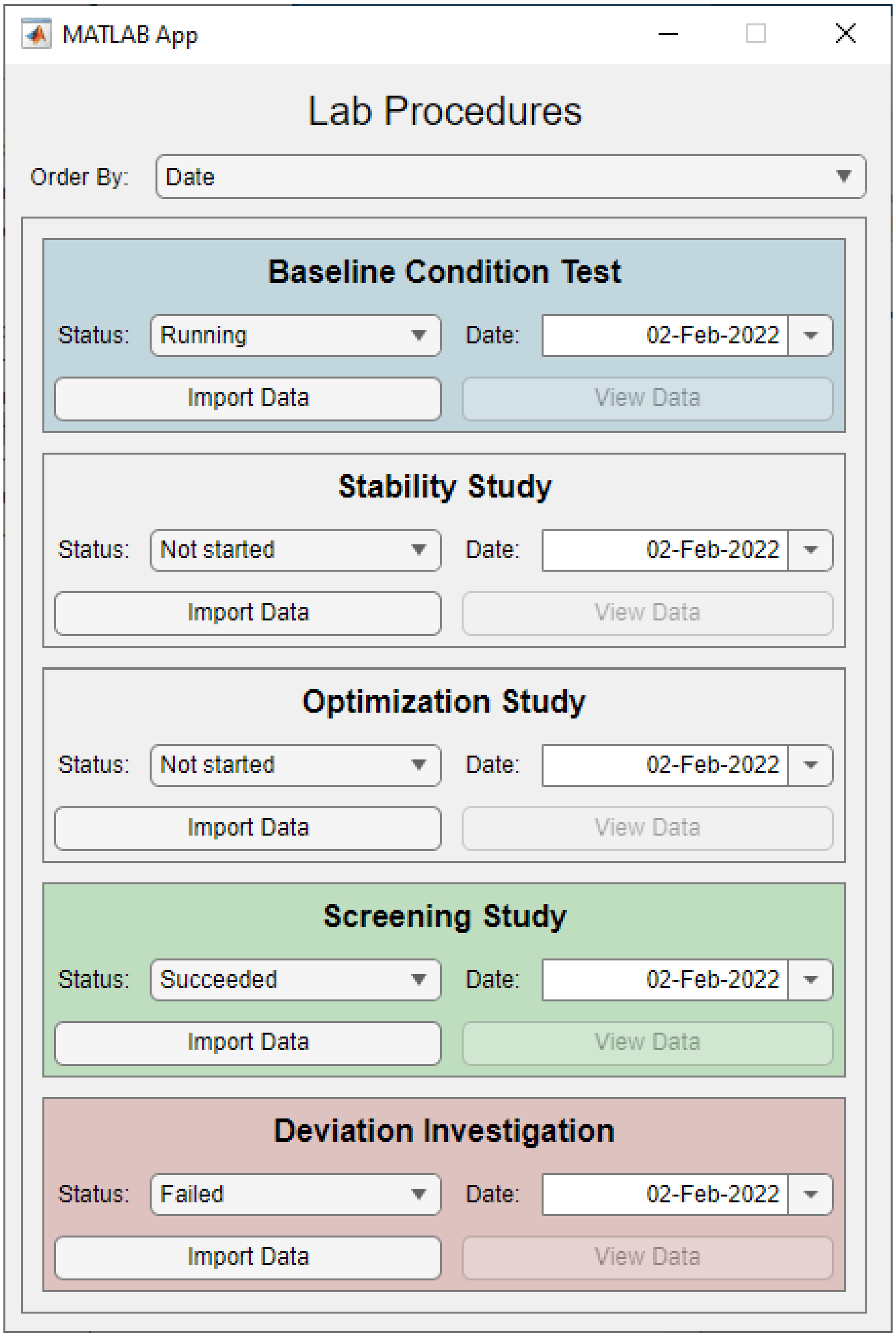
Some benefits of this method include:

- **Reusability** — You can reuse a custom UI component within a single app or across multiple apps with minimal effort.
- **Maintainability** — You can reduce duplicate code by componentizing pieces of your app layout that perform similar functions.
- **Scalability** — You can more easily extend app functionality when your code is organized into multiple self-contained portions.

Starting in R2022a, you can use App Designer to create custom UI components, and then use those components in your apps. For more information, see “Create a Simple Custom UI Component in App Designer” on page 13-2.

Example Overview

This example explores an app that allows users to store and modify information about their lab procedures. An app user can update the status or date of a procedure, import data associated with a procedure, and order the procedures based on their titles, statuses, or dates.



Lab Procedures

Order By: Date

Baseline Condition Test
Status: Running Date: 02-Feb-2022

Stability Study
Status: Not started Date: 02-Feb-2022

Optimization Study
Status: Not started Date: 02-Feb-2022

Screening Study
Status: Succeeded Date: 02-Feb-2022

Deviation Investigation
Status: Failed Date: 02-Feb-2022

The example includes two different ways to create the app in App Designer:

1. As a self-contained app with all layout and behavior code contained in a single app file

2. As a modular app where each lab procedure in the app is represented by a `LabProcedure` object, which is created as a custom UI component in App Designer

The example contains these files:

- `LabProcedureApp_WithoutComponent.mlapp` — Self-contained app file
- `LabProcedure.mlapp` — Custom UI component file
- `LabProcedureApp_WithComponent.mlapp` — Modular app file

These steps describe how to take the self-contained app and make it more modular by creating and using the `LabProcedure` custom UI component.

Create the Custom UI Component

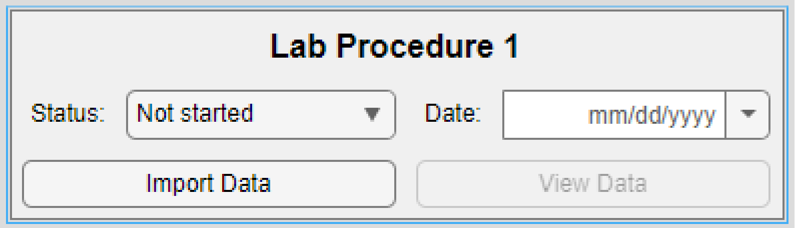
To modularize an app by creating a custom UI component, first identify the portions of your app that can be extracted to a separate component file. For example, there are certain characteristics that every lab procedure in the app has in common. These aspects are captured in the `LabProcedure` custom UI component.

To explore the component code, open the `LabProcedure.mlapp` file in App Designer.

Lay Out Component

In **Design View**, lay out the `LabProcedure` component by adding the features that every lab procedure in the app has in common:

- A label for the lab procedure title
- A status drop-down component with the options `Not started`, `Running`, `Succeeded`, and `Failed`, and a status label
- A date picker component and a date label
- A button to import data
- A button to view data



Program Underlying Component Behavior

In **Code View**, program the behavior that every lab procedure in the app has in common by adding underlying component callbacks. For example, add a `ButtonPushedFcn` callback to the **Import Data** button that allows app users to select a file when they click the button.

Create Public Properties

The app needs access to certain information about each lab procedure. For example, to order the procedures by status, the app needs access to the statuses. Provide this access by creating public properties.

Create these public properties for the component:

- `Title`

- Status
- Date

Then, write code to associate the properties with the LabProcedure component appearance and behavior.

For more information, see “Create Public Properties for Custom UI Components in App Designer” on page 13-13.

Create Public Callbacks

The app needs to execute a response when a user interacts with a lab procedure. For example, to update the background color of the procedure based on its status, the app needs to execute a response to an app user changing the status in the drop-down list. Provide the ability for an app creator to program a response to an interaction in the context of the app by creating event–public callback pairs.

Create events with these associated public callbacks for the LabProcedure component:

- StatusChangedFcn
- DateChangedFcn

Then, write code to trigger the event and execute the callback when the drop-down component value changes.

For more information, see “Create Callbacks for Custom UI Components in App Designer” on page 13-21.

Configure Component for Use in Apps

To add the LabProcedure component to the **Component Library**, click the **Configure for Apps** button in the **Designer** tab and fill out the App Designer Custom UI Component Metadata dialog box.

For more information, see “Configure Custom UI Components for App Designer” on page 12-17.

Use the Custom UI Component in Your App

After you create and configure your custom UI component, incorporate the component into your app.

To explore the app code that uses the LabProcedure component, open the LabProcedureApp_WithComponent.mlapp file in App Designer.

Lay Out App

Replace the portions of the app layout that represent a lab procedure with LabProcedure components. Under the **Example Components (Custom)** section of the **Component Library**, drag the LabProcedure components onto the app canvas.

Use the Property Inspector to customize the appearance of each of the procedures in the app. For example, update the Title property of each lab procedure.

Update App Code

Update your app code to refer to the LabProcedure components and to query and set their public properties when needed.

For example, the original example app contains a helper function named updateBackgroundColor. Update this helper function code to change the background color of the LabProcedure component when its status changes. Because the LabProcedure component has a Status public property and an inherited BackgroundColor public property, the updated helper function code is simple and easy to read:

```
function updateBackgroundColor(app,lp)
    switch lp.Status
        case "Not started"
            lp.BackgroundColor = [0.94 0.94 0.94];
        case "Running"
            lp.BackgroundColor = [0.76 0.84 0.87];
        case "Succeeded"
            lp.BackgroundColor = [0.75 0.87 0.75];
        case "Failed"
            lp.BackgroundColor = [0.87 0.76 0.75];
    end
end
```

Add Component Callbacks

Add a StatusChangedFcn callback to each LabProcedure component by right-clicking the component and selecting **Callbacks > Add StatusChangedFcn callback**. Call the updateBackgroundColor and updateProcedureOrder helper functions to update the app when an app user changes the status of a lab procedure.

To run the app, click **Run**.

See Also

Related Examples

- “Organize App Data Using MATLAB Classes” on page 8-2
- “Create Public Properties for Custom UI Components in App Designer” on page 13-13
- “Create Callbacks for Custom UI Components in App Designer” on page 13-21

Verify Behavior of Custom UI Components in App Designer

Starting in R2022a, you can create custom UI components interactively in App Designer. For an example of how to use App Designer to create a slider-spinner component with linked values, see “Create a Simple Custom UI Component in App Designer” on page 13-2.

While you are developing your custom UI component in App Designer, there are multiple ways to verify that your component works as you expect.

- “Run Component” on page 13-36 — Use this method to run your code to check for errors and to view the component in a UI figure window.
- “Create Component from Command Window” on page 13-36 — Use this method to verify the public properties and callbacks of your component.
- “Add Component to App” on page 13-38 — Use this method only when you are done developing and debugging your component code, to view your component in an App Designer app.

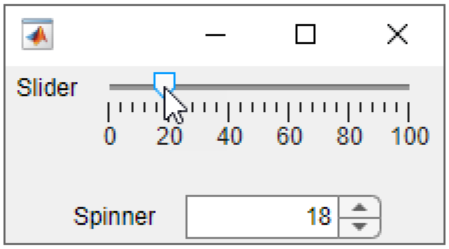
Run Component

At any point in the development process, you can run your component code by clicking ▶ **Run**. App Designer creates a UI figure window that contains your custom UI component.

Run your component while you are actively developing your custom UI component to:

- Check that your component is created without errors.
- Debug your component code using the debugger.
- View your component layout in a running app.
- Verify the behavior of underlying component callbacks.

For example, while you are developing a slider-spinner component with linked values, you can run the component to verify that moving the slider thumb also changes the spinner value.



Create Component from Command Window

When your component code runs without errors and you are ready to verify the behavior of public properties and callbacks, create the component from the MATLAB Command Window. Use this method to:

- Verify how your component responds when a public property is set.
- Assign a callback to your component and verify that it executes in response to an interaction.

To create the component from the Command Window, add the folder that contains your component MLAPP file to the MATLAB path. Enter the component name and return the component object as a variable:

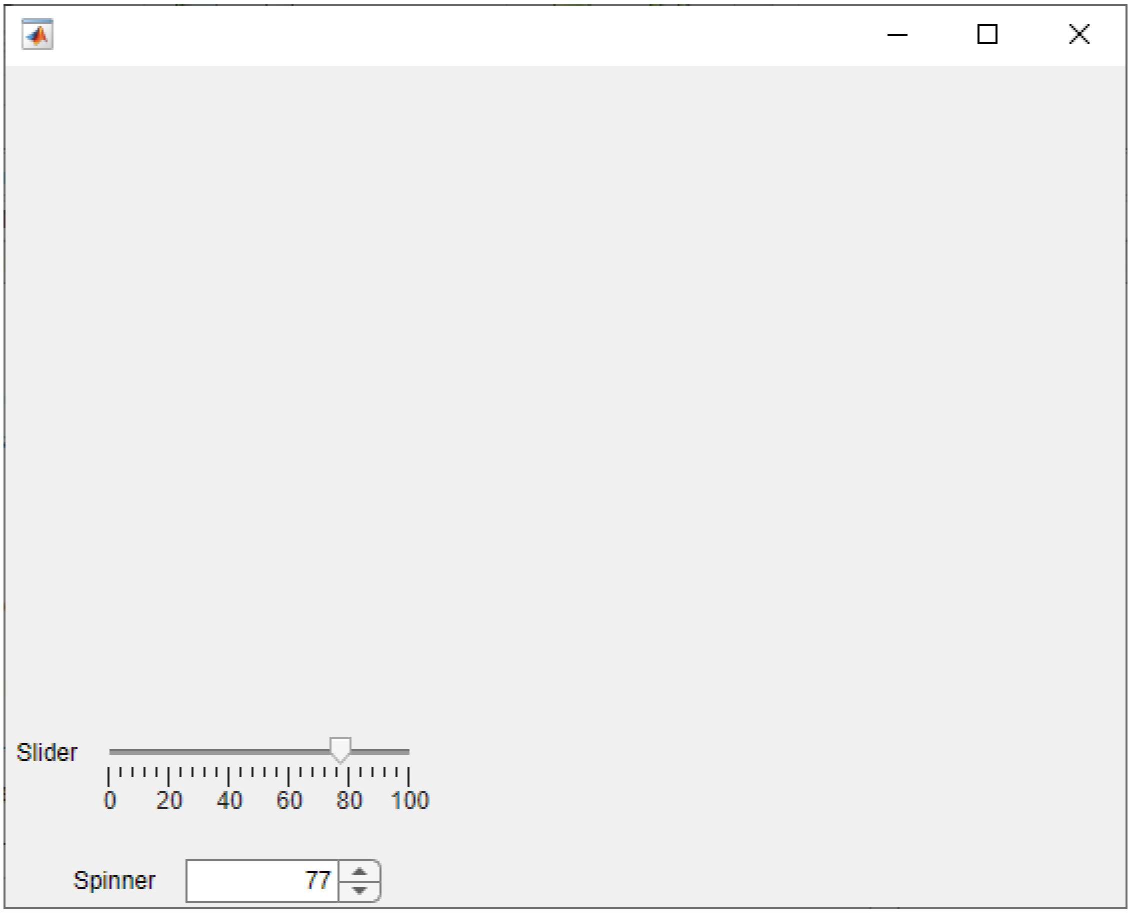
```
comp = ComponentFileName
```

You can then set public properties and assign callbacks to the component object.

For example, if you have created a slider-spinner component by following the steps in “Create a Simple Custom UI Component in App Designer” on page 13-2 and saved the component file as `SliderSpinner.mlapp`, you can verify the property and callback behavior programmatically.

To set public properties during component creation, pass the properties as name-value arguments. Create the component and set the `Value` property using a name-value argument:

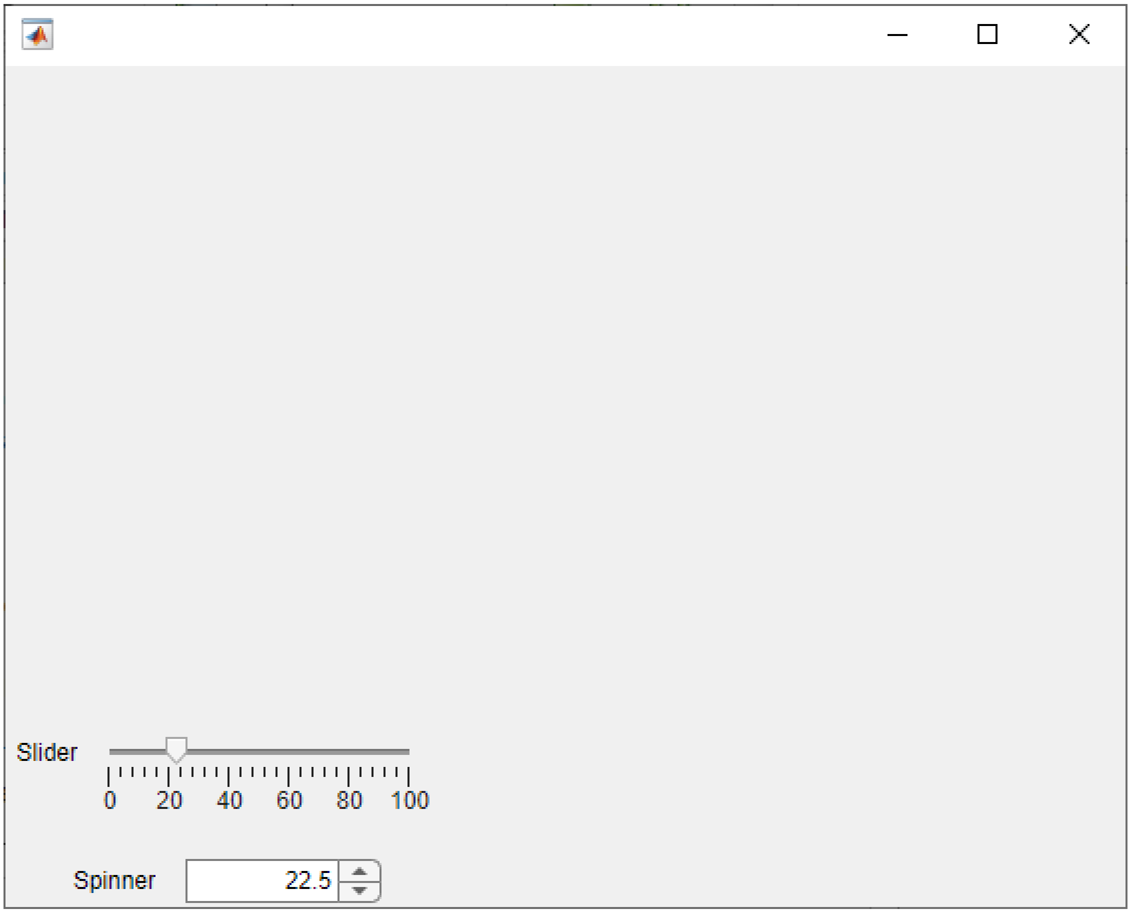
```
comp = SliderSpinner(Value=77);
```



Spinner 77

To set the value of public properties after component creation, use dot notation. Change the `Value` property of the `comp` object, and verify that the slider and spinner components update to reflect the new value:

```
comp.Value = 22.5;
```



Assign a `ValueChangedFcn` callback that displays the value in the MATLAB Command Window.

```
comp.ValueChangedFcn = @(src,event)disp(src.Value);
```

Move the thumb on the slider and change the spinner value to verify the callback behavior.

Add Component to App

Once you are done debugging and verifying the behavior of specifying properties and callbacks for your component, you can add the component to an App Designer app. Use this method when the development of your component is complete to view the component from the point of view of an app creator who uses the component.

Note Avoid using your component in an App Designer app to debug and verify the component behavior while you are actively developing your component. If you make a change to your component code while the component is being used in an open App Designer app, you must restart App Designer to see the changes reflected within the app.

To view your component in an App Designer app, open the component file in App Designer and follow these steps:

1. In the **Designer** tab, click **Configure for Apps**.
2. Fill out the App Designer Custom UI Component Metadata dialog box, and then click **OK**.
3. In the confirmation dialog box, click **Add to Path** to add the component and generated `resources` folder to the MATLAB path.
4. In the **Designer** tab, click **New** and select **Blank App**.
5. Drag the component from the **Component Library** onto the app canvas.

For more information, see “Configure Custom UI Components for App Designer” on page 12-17.

See Also

Related Examples

- “Create a Simple Custom UI Component in App Designer” on page 13-2
- “Debug MATLAB Code Files”

Create Custom UI Component with a Chart in App Designer

This example shows how to incorporate a chart into your custom UI component. App creators can use this component to let app users interactively change the display of bubble chart data. The component interface consists of:

- Public properties to specify data to plot in the bubble chart, such as `XData`, `YData`, and `SizeData`
- Public properties to configure the bubble chart display, such as `XLabel`, `YLabel`, and `ShowLegend`
- Public properties to configure the user interface controls, such as `ShowSizeUI` and `ShowTransparencyUI`
- Public callbacks `BubbleSizeChangedFcn` and `BubbleTransparencyChangedFcn` that execute when an app user interacts with the size and transparency controls in an app

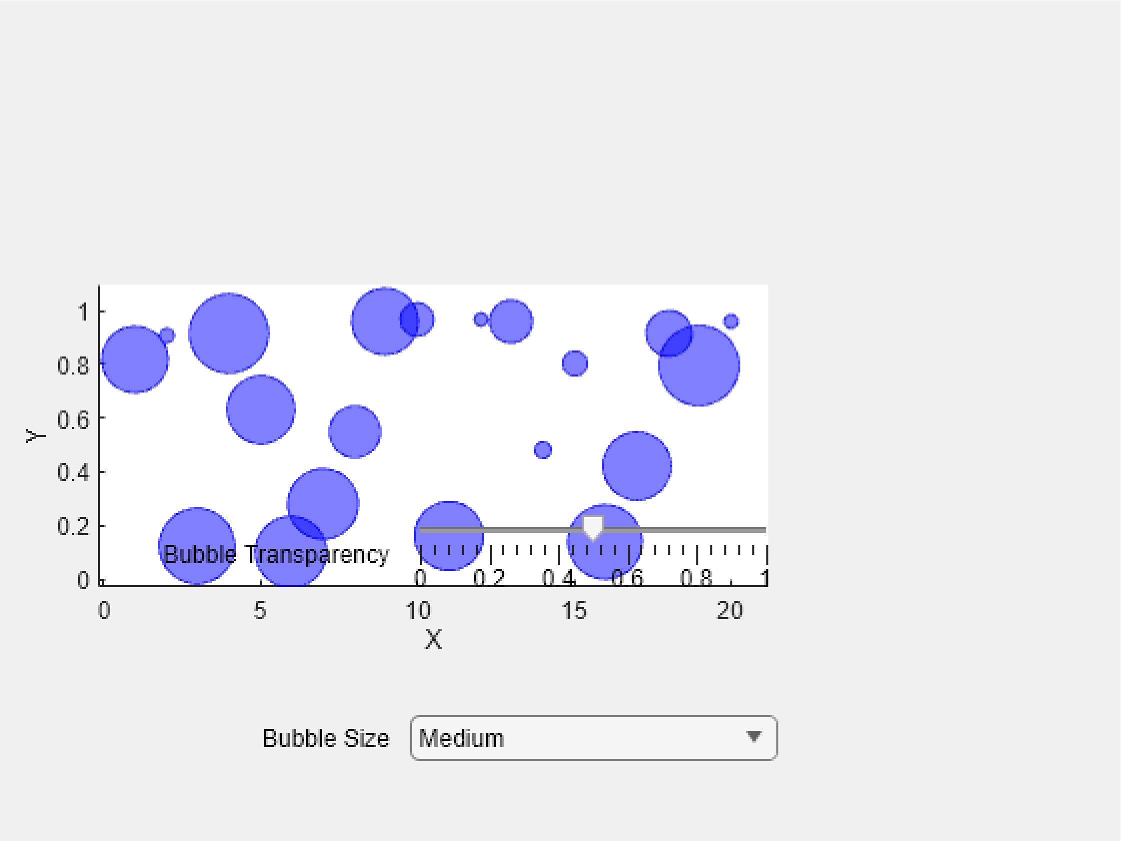
To explore the custom component code, open the `InteractiveBubbleChart.mlapp` file in App Designer.

To verify the custom component behavior, first open a new script file and create some random data to plot.

```
x = 1:20;
y = rand(1,20);
bsize = rand(1,20);
```

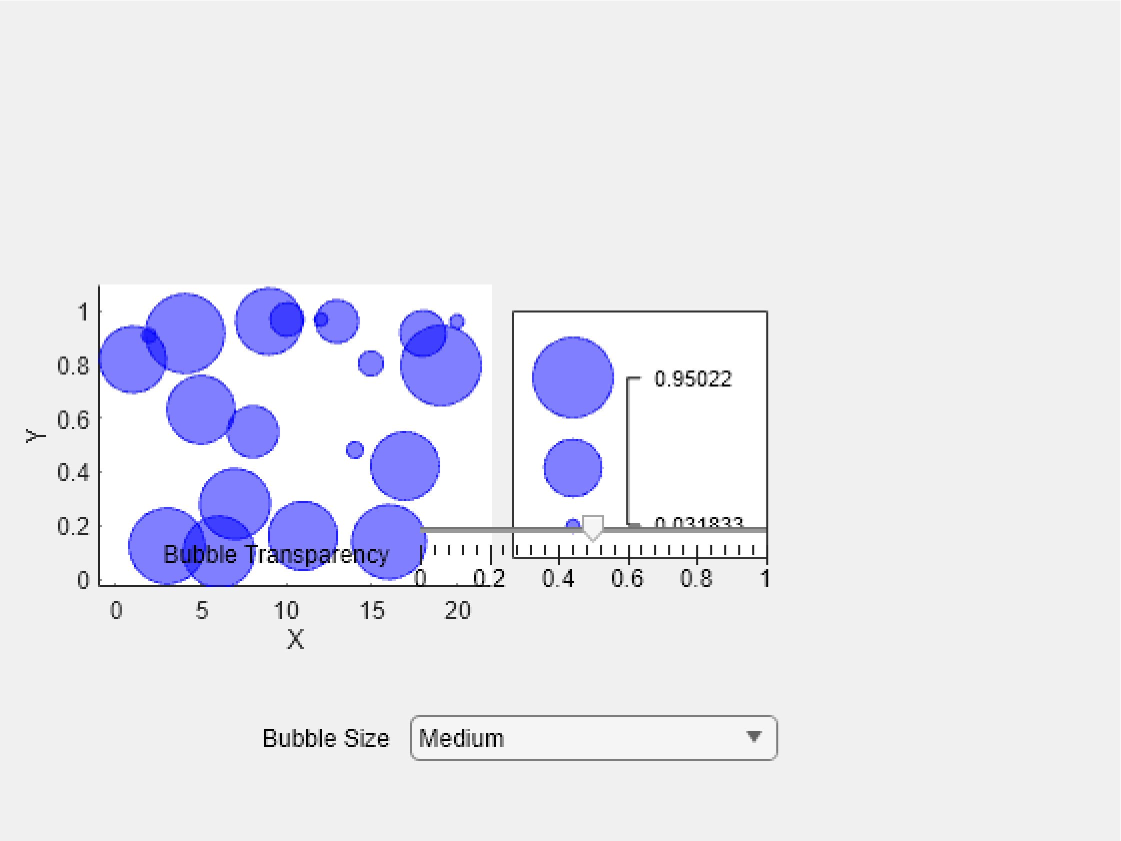
Add this code to create an `InteractiveBubbleChart` object programmatically and specify the data-related public properties as name-value arguments.

```
comp = InteractiveBubbleChart(XData=x,YData=y,SizeData=bsize);
```



Update the component to show a legend for the plot, and define a callback function that hides the legend when the bubble sizes are large by adding this code to the file. Run the code and interact with the component to see the behavior.

```
comp.ShowLegend = true;
comp.BubbleSizeChangedFcn = @toggleLegend;

function toggleLegend(src,event)
    if src.BubbleSize == "Large"
        src.ShowLegend = false;
    else
        src.ShowLegend = true;
    end
end
```



Bubble Size Medium

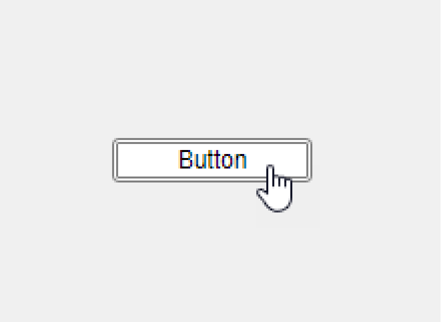
See Also

Related Examples

- “Create a Simple Custom UI Component in App Designer” on page 13-2
- “Verify Behavior of Custom UI Components in App Designer” on page 13-36

Create Custom Button with Hover Effect Using HTML

This example shows how to create a custom button component with a hover effect in App Designer by using an HTML UI component.



App creators can add this button to their apps and customize its appearance and behavior. The component interface consists of:

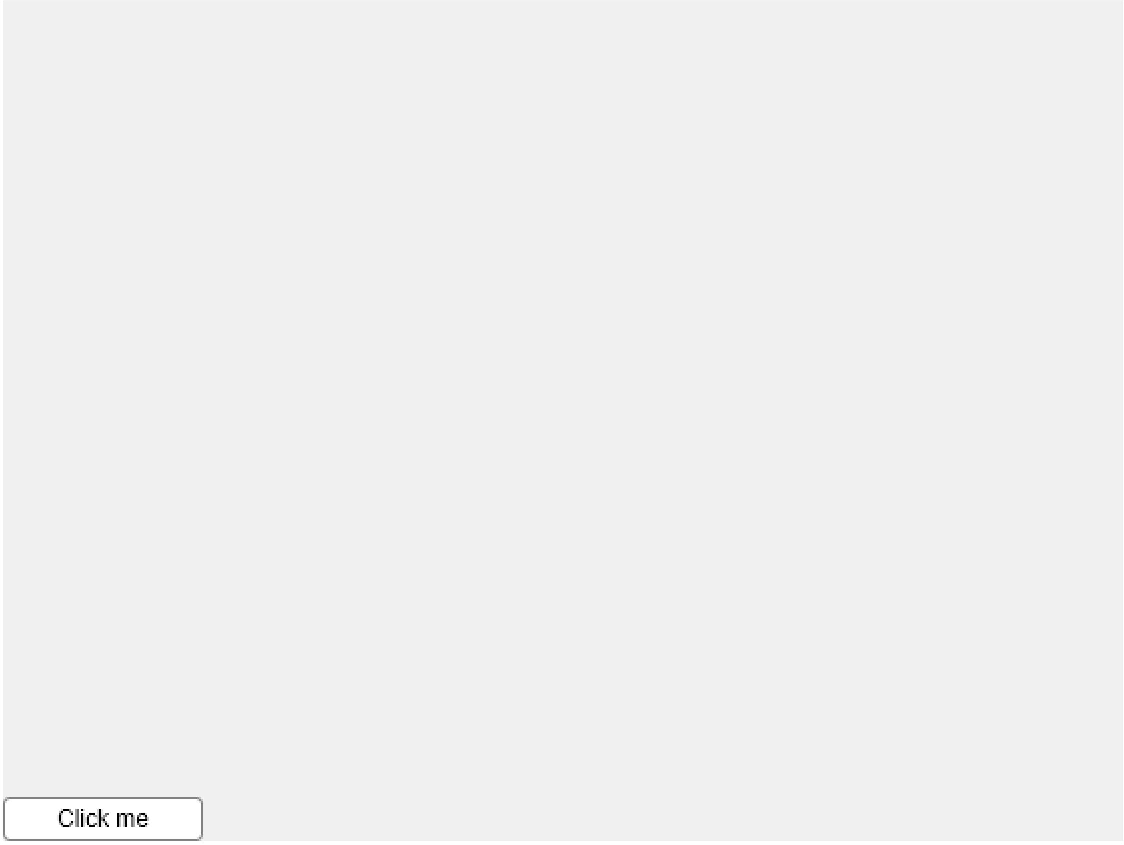
- Public properties `FontColor` and `Text` to customize the button appearance
- Public callback `ButtonPushedFcn` that executes when an app user pushes the button in an app

The `HoverButton` component appearance and hover effect are defined using HTML in the `Button.html` source file. The component interface and behavior, including its public properties and callback, are defined in the `HoverButton.mlapp` file in App Designer.

The underlying component in the `HoverButton.mlapp` file is an HTML UI component with its `HTMLSource` property specified as the `Button.html` file. The two files communicate with each other using the `Data` property and `DataChanged` event, which are synchronized between the HTML UI component in MATLAB® and the `htmlComponent` JavaScript® object in the HTML source file.

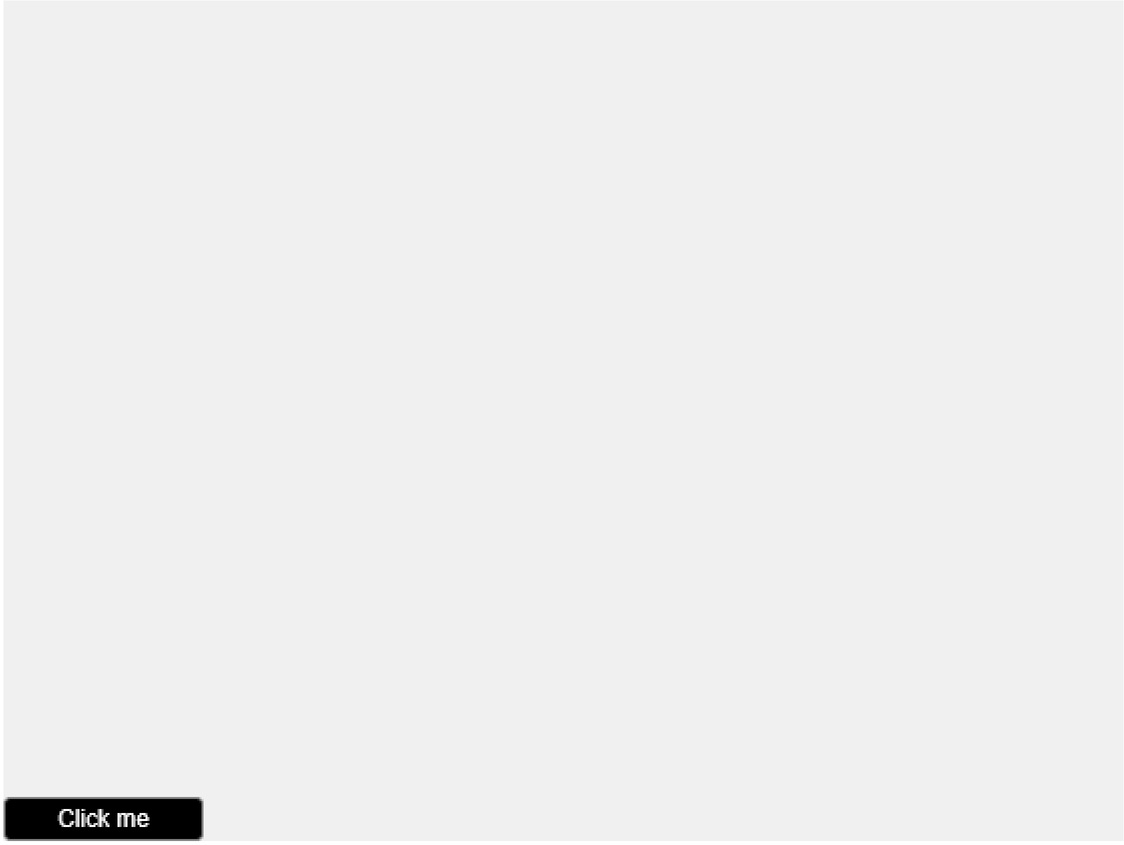
To verify the custom component behavior, create a `HoverButton` object programmatically in the MATLAB Command Window. Modify the default button text by specifying the `Text` property as a name-value argument. Hover over the button to see the hover effect.

```
comp = HoverButton(Text="Click me");
```



Update the button color to black and the font color to white.

```
comp.BackgroundColor = "black";
comp.FontColor = "white";
```



Create a callback function that displays text in the Command Window when a user pushes the button.

```
comp.ButtonPushedFcn = @(src,event)disp("Button pushed");
```

Click the button. The text "Button pushed" displays.

See Also

uihtml

Related Examples

- “Create a Simple Custom UI Component in App Designer” on page 13-2
- “Create Custom UI Component With HTML” on page 12-24
- “Create HTML Content in Apps” on page 4-23

Create Event Data for Custom UI Component Callbacks

Every UI component callback has associated event data. This data provides information about the specific user interaction associated with the callback. MATLAB automatically passes this event data as the second argument to any callback function. For example, when you specify a `ValueChangedFcn` callback function for a slider UI component, MATLAB passes the event data as the second argument to the callback function. The event data has properties that provide specific information about the user interaction, including a `Value` and `PreviousValue` property. This information is often useful for app authors to access when programming a response to a user interaction.

When you create a custom UI component with a public callback, that callback has some default associated event data. However, you can also define your own custom event data to provide additional information about the user interaction that executes the callback.

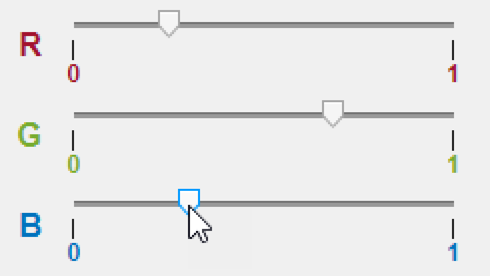
View Default Event Data

When you create an event–public callback pair for a custom UI component, the event data associated with the pair has two properties:

- `Source` — The component that executes the callback
- `EventName` — The name of the event associated with the callback

For more information about creating event–public callback pairs, see “Create Callbacks for Custom UI Components in App Designer” on page 13-21.

For example, in App Designer, create a custom RGB picker UI component that consists of three sliders to choose red, green, and blue values to define a color. Save the component as `RGBPicker.mlapp`.



Create an event named `ColorChanged` with associated public callback `ColorChangedFcn`. Trigger the event to execute the callback whenever a user changes the value of one of the sliders by adding this code to the `ValueChangedFcn` callback of the underlying slider components.

```
notify(comp,"ColorChanged")
```

This code executes the `ColorChangedFcn` callback and passes it the default event data. To view the event data, create a `ColorChangedFcn` callback that displays the event data by typing these commands in the MATLAB Command Window.

```
comp = RGBPicker;
comp.ColorChangedFcn = @(src,event)disp(event);
```

Adjust one of the sliders. The event data displays in the Command Window.

```
  EventData with properties:

       Source: [1×1 RGBPicker]
    EventName: 'ColorChanged'
```

For more information about the default event data, see `event.EventData`.

Define Custom Event Data

Define custom event data when you want to provide additional information about a user interaction to a callback function. For example, an app creator who uses the RGB picker component might want to access the new and previous selected color when an app user interacts with the component.

To customize event data, create a new class as a subclass of the `event.EventData` class and define any additional event data properties. For example, to create custom event data for the RGB picker component, create a new file named `RGBEventData.m` and save it in the same folder as the custom RGB picker component file. Add code to the file to:

- Define a new class named `RGBEventData`.
- Define two properties to store the previous RGB triple and the new RGB triple.
- Define the class constructor to take in the previous and new RGB triples and assign the values to the properties.

```
classdef RGBEventData < event.EventData
    properties
        PreviousRGB
        RGB
    end

    methods
        function eventData = RGBEventData(prevRGB,newRGB)
            eventData.PreviousRGB = prevRGB;
            eventData.RGB = newRGB;
        end
    end
end
```

To create and use this event data for the custom UI component, add code to the `RGBPicker.mlapp` file to:

- Create a public property for the component named `RGB` that stores the slider values as an RGB triple. For more information, see “Create Public Properties for Custom UI Components in App Designer” on page 13-13.
- In the `ValueChangedFcn` callback of the underlying slider components, query the previous RGB value and the new RGB value, and use them to create the event data.
- Trigger the `ColorChanged` event to execute the `ColorChangedFcn` callback with the custom event data by calling the `notify` function.

```
prevRGB = comp.RGB;
newRGB = [comp.RSlider.Value comp.GSlider.Value comp.BSlider.Value];
comp.RGB = newRGB;
eventData = RGBEventData(prevRGB,newRGB);
notify(comp,"ColorChanged",eventData);
```

To view the full RGBPicker component code in App Designer, open the example in “Example: UI Component with Custom Event Data” on page 13-48.

To view the event data, create a ColorChangedFcn callback that displays the event data by typing these commands in the MATLAB Command Window.

```
comp = RGBPicker;
comp.ColorChangedFcn = @(src,event)disp(event);
```

Adjust one of the sliders. The custom event data displays in the Command Window.

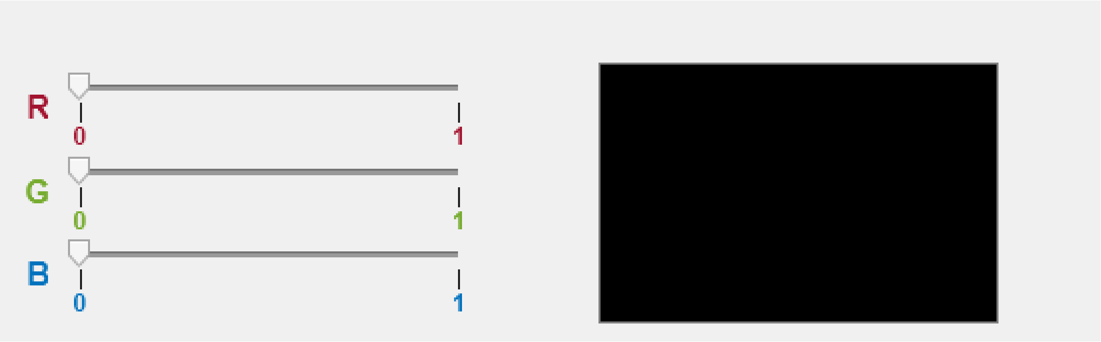
```
  RGBEventData with properties:

    PreviousRGB: [0 0 0]
            RGB: [0.5630 0 0]
         Source: [1×1 RGBPicker]
      EventName: 'ColorChanged'
```

Example: UI Component with Custom Event Data

Use the custom event data of the RGBPicker UI component to update the background color of a panel when the slider values changes.

- Create an RGBPicker component and a panel in a UI figure.
- Define a ColorChangedFcn callback for the component that executes when a user updates one of the slider values.
- In the callback function, use the event input argument to access the new RGB triple value associated with the user interaction and update the panel background color.

```
fig = uifigure;
fig.Position(3:4) = [550 170];
c = RGBPicker(fig);
p = uipanel(fig);
p.Position = [300 10 200 130];
p.BackgroundColor = c.RGB;
c.ColorChangedFcn = @(src,event)updatePanelColor(src,event,p);

function updatePanelColor(src,event,p)
color = event.RGB;
p.BackgroundColor = color;
end
```



See Also

Related Examples

- “Create Callbacks for Custom UI Components in App Designer” on page 13-21
- “Events and Listeners Syntax”

Configure Property Display for Custom UI Components in App Designer

When you develop a custom UI component in App Designer, you can provide customization options for app creators that use your component by creating public properties. This example shows how to design these public properties so app creators can easily modify them from within App Designer. Specify the data types and other requirements for the public properties so app creators can enter only valid values. For more information about creating public properties, see "Create Public Properties for Custom UI Components in App Designer" on page 13-13.

This table shows how each of the property types appears in the **Component Browser** when the component is used in an app.

Property Type	Component Browser Input
Numeric	Edit field
Text	Edit field
Logical	Check box
Enumeration	Drop-down list

Create Custom UI Component for App Text Styling

For this example, create a custom component to quickly and consistently style app text, such as changing the font size and color of the text in the app. Create a component named `StylableText` that consists of an underlying label component and four public properties to demonstrate the allowable property types for a custom UI component:

- `Scale`: Font size scale, specified as a numeric property of type `double`. App creators can change the property in an App Designer app by entering a value greater than `0` in an edit field.
- `Text`: Component text, specified as a text property of type `string`. App creators can change the property in an App Designer app by entering text in an edit field.
- `WordWrap`: Text wrap, specified as a logical property of type `matlab.lang.OnOffSwitchState`. App creators can change the property in an App Designer app by toggling a check box.
- `Style`: Font size and color, specified as an enumeration property of type `TextStyle` (custom enumeration class). App creators can change the property in an App Designer app by selecting an option from a drop-down list.

To create the custom component, open a new blank component in App Designer. In **Design View**, add a **Label** from the **Component Library**. Save this new component as `StylableText.mlapp`.

Create Numeric Property

Create a public property with a numeric data type when a component has a numeric value that app creators can change.

In the `StylableText` component, create a public numeric property that scales the font size of each style given by the `Style` property. Specify these fields in the **Add Property** dialog box:

- **Name**: `Scale`

- **Data Type**: double
- **Default Value**: 1
- **Validation Functions**: mustBeGreaterThan(0)

Add this code to the update function in **Code View** to set the FontSize property of the underlying label component. Define FontSize as a product of the font size given by the Style property and Scale.

```
comp.Label.FontSize = comp.Style.FontSize*comp.Scale;
```

When app creators use the component in an app, they can change the text scaling by entering a scale factor greater than 0 in the Scale edit field in the **Component Browser**.

Create Text Property

Create a public property with a textual data type when a component has text that app creators can change.

In the StylableText component, create a public text property that controls the text of the label component. Specify these fields in the **Add Property** dialog box:

- **Name**: Text
- **Data Type**: string
- **Default Value**: "Example"

Add this code to the update function in **Code View** to link the public property to the Text property of the underlying label component.

```
comp.Label.Text = comp.Text;
```

When app creators use the component in an app, they can change the label text by entering text in the Text edit field in the **Component Browser**.

Create On/Off Logical Property

Create a public property with a logical data type when a component has two distinct states that app creators can change. Use the matlab.lang.OnOffSwitchState data type. Existing UI components in MATLAB use this data type for logical properties. Type this class into the **Data Type** field instead of selecting an option from the drop-down list.

In the StylableText component, create a public logical property that determines whether the text wraps. Specify these fields in the **Add Property** dialog box:

- **Name**: WordWrap
- **Data Type**: matlab.lang.OnOffSwitchState
- **Default Value**: matlab.lang.OnOffSwitchState.on

Add this code to the update function in **Code View** to link the public property to the WordWrap property of the underlying label component.

```
comp.Label.WordWrap = comp.WordWrap;
```

When app creators use the component in an app, they can control whether the text wraps with the WordWrap check box in the **Component Browser**.

Create Custom Enumerated Property

Create a public property with a custom enumeration class when a component has discrete predetermined configurations that app creators can select. First, define an enumeration class in MATLAB that stores the valid public property values and configurations. To define an enumeration class, add an enumeration block to a class definition. Then, add a public property with that enumeration class as the data type to the component in App Designer. For more information on enumeration classes, see “Define Enumeration Classes”.

For this example, create a class that enumerates a set of formatting styles, such as title and heading styles. Each style has associated values that specify the font size and color. In the MATLAB Editor:

- Create a new class named TextStyle.
- Define two properties to store the font size and color.
- Define an enumerated set of text styles with associated properties.

```
classdef TextStyle
    properties
        FontSize
        FontColor
    end

    enumeration
        Body (12, [0 0 0])
        Heading1 (18, [0 0.4470 0.7410])
        Heading2 (16, [0.3010 0.7450 0.9330])
        Title (24, [0.6350 0.0780 0.1840])
    end

    methods
        function obj = TextStyle(size,color)
            obj.FontSize = size;
            obj.FontColor = color;
        end
    end
end
```

Save the class as TextStyle.m in the same folder as the StylableText component.

In the StylableText component in App Designer, create a public property. Specify these fields in the **Add Property** dialog box:

- **Name**: Style
- **Data Type**: TextStyle
- **Default Value**: TextStyle.Title

Add this code to the update function in **Code View** to link the public property to the FontSize and FontColor properties of the underlying label component.

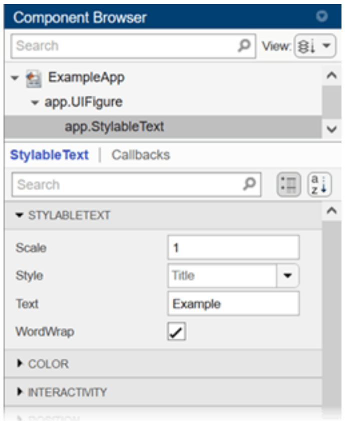
```
comp.Label.FontSize = comp.Style.FontSize*comp.Scale;
comp.Label.FontColor = comp.Style.FontColor;
```

When app creators use the component in an app, they can modify the formatting of the text by selecting a different style option from the `Style` drop-down list in the **Component Browser**. For example, setting the `Style` property to `Title` sets the label font size to 24 and the label font color to the RGB triplet `[0.6350 0.0780 0.1840]`.

Use Custom UI Component in App

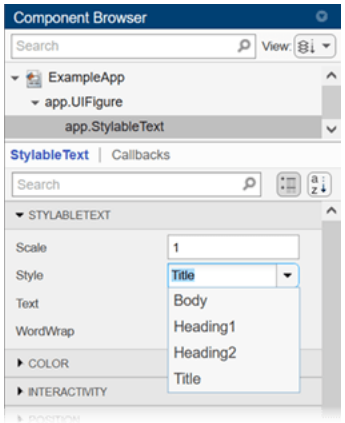
To see how the public properties of the `StylableText` component appear in the **Component Browser**, configure the component for use in an app. For more information, see "Configure Custom UI Components for App Designer" on page 12-17.

In a new blank app in App Designer, add a `StylableText` component. Edit the public properties in the **Component Browser** using the different input fields.



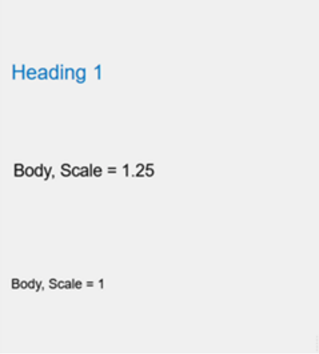
Scale 1
Style Title
Text Example
WordWrap ☑

Use the drop-down list to select a different option for the `Style` property. The options are defined by the enumeration in the `TextStyle` class.



Scale 1
Style Title
- Body
- Heading1
- Heading2
- Title

Use multiple `StylableText` components with different property values in the same app. For example, this app has three components. One uses the `Style` option `Heading1`. The other two use the `Style` option `Body`. One of these components sets `Scale` to `1`. The other sets `Scale` to `1.25`. The value of the `Text` property of each component describes the corresponding text style.



Heading 1

Body, Scale = 1.25

Body, Scale = 1

See Also

`matlab.lang.OnOffSwitchState`

Related Examples

- “Create a Simple Custom UI Component in App Designer” on page 13-2
- “Create Public Properties for Custom UI Components in App Designer” on page 13-13
- “Configure Custom UI Components for App Designer” on page 12-17
- “Define Enumeration Classes”

Transition or Maintain `figure`-Based Apps

Update figure-Based Apps to Use uifigure

In this section...

MATLAB provides two functions to create a figure window: `figure` and `uifigure`. While both of these functions create a `Figure` object, there are some differences in the way that this object is configured and the capabilities it supports. Figures created using the `uifigure` function are configured primarily for app building, whereas figures created using the `figure` function are configured primarily for data exploration and visualization.

The `uifigure` function is the recommended function to use when building new apps programmatically, and is the function that App Designer uses to create apps. The `figure` function will continue to be supported, but there are many new app building capabilities that can be used only with UI figures. This page provides an overview of the differences between `Figure` objects created using the `figure` function and the `uifigure` function, and information about how to update your app to take advantage of the `uifigure`-based app building capabilities.

Overview for Updating Your App

To update your `figure`-based app to use `uifigure` and take advantage of the additional capabilities in UI figures, follow these steps:

1. "Update App Figure and Containers" on page 14-7 — Replace calls to `figure` with `uifigure`, and update the properties of the `Figure` object and other app containers, such as `Panel` and `TabGroup` objects.
2. "Update UIControl Objects and Callbacks" on page 14-11 — Replace calls to `uicontrol` with analogous UI component functions, and update component properties and callbacks.
3. "Update Dialog Boxes" on page 14-18 — Replace calls to dialog box functions such as `errordlg` and `warndlg` with dialog box functions configured for app building such as `uialert`.

Capabilities Only Available with UI Figures

Some benefits of updating your app to use the `uifigure` function include:

- **Additional component types** — UI figures support additional modern app building components, such as:
  - Trees
  - Spinners
  - Hyperlinks
  - Instrumentation components such as gauges and switches
  - HTML UI components that let you embed third-party visualizations in your app
- **Modern layout and resize options** — UI figures support grid layout managers and component auto-resize behavior as an alternative to manually specifying the `Position` property and writing resize code in a `SizeChangedFcn` callback. Using these alternatives can greatly simplify app layout code.

- **Additional capabilities for existing components** — Components in UI figures support additional customization options, including:
  - Making containers scrollable
  - Styling individual table cells to change the color and font, and to add icons and format text
  - Displaying `table` array data in table UI components

To see a list of all components supported in `uifigure`-based apps, see "App Building Components" on page 4-2.

Differences Between figure-Based and uifigure-Based Apps

The major differences between `figure`-based apps and `uifigure`-based apps are due to differences in the underlying `Figure` object configuration and unsupported functionality. Understanding these differences will help you update your `figure`-based app to use `uifigure`.

Differences in Default Configuration

Because figures created using the `uifigure` function are configured for app building instead of data exploration, there are some differences in the default configuration of those `Figure` objects when compared to figures created using the `figure` function. This table lists the major differences.

Category	`figure` Configuration	`uifigure` Configuration	Explanation of Difference
Menu and toolbar	The figure window has a default menu and toolbar with common data exploration functionality.	The UI figure window does not have a default menu and toolbar.	The functionality that the menu and toolbar provide is less relevant for app building than for data exploration. You can create your own custom menu and toolbar for the apps you create by using the `uimenu` and `uitoolbar` functions.

Category	figure Configuration	uifigure Configuration	Explanation of Difference
HandleVisibility value	The HandleVisibility of the figure is 'on' by default.	The HandleVisibility of the UI figure is 'off' by default.	The value of the HandleVisibility property controls whether the figure or the objects it contains can become the current object (for example, using gcf or gca). Many graphics functions implicitly use gcf or gca to determine the target for operations such as plotting data. The HandleVisibility of a UI figure is 'off' by default so that functions do not make unwanted changes to the user interface.
Resize behavior	Resizing the figure window has no effect on the size of controls and containers such as UIControl, Table, and Panel objects by default.	The UI figure has a property named AutoResizeChildren that is set to 'on' by default. When AutoResizeChildren is 'on', MATLAB automatically resizes objects in the UI figure window whenever the UI figure window is resized. You can set AutoResizeChildren to 'off' to disable this resize behavior.	Resizing UI components when a user resizes a UI figure window enables app use at any window size. The auto-resize behavior in UI figures provides a lightweight default behavior in addition to other resize management options such as grid layout managers.

Category	`figure` Configuration	`uifigure` Configuration	Explanation of Difference
Container location, size, and units	By default, `Panel`, `ButtonGroup`, and `TabGroup` objects parented to the figure have `Units` set to `'normalized'` and occupy the full size of the figure window.	All containers and UI components parented to the UI figure have a set default location and size, specified in pixel units.	Using pixel units to manually specify the position of containers and UI components provides the most control over your app layout. If you want to automatically resize containers or components based on the size of their parent in a UI figure, create a grid layout manager using the `uigridlayout` function.

Unsupported Functionality in UI Figures

As of R2023a, some functionality that is supported in figures created using the `figure` function is not supported in figures created using the `uifigure` function. This table lists common scenarios and coding patterns that require extra steps or manual code changes when updating your apps to use `uifigure`.

Category	Not Supported	Suggested Actions
Controls created using `uicontrol`	User interface controls created using the `uicontrol` function are not supported in figures created using the `uifigure` function.	Update your app to use the corresponding function to create a UI component in a `uifigure`-based app. For example, use the `uibutton` function to create a push button. For more information, see “Update UIControl Objects and Callbacks” on page 14-11.
Menu and toolbar	Figures created using the `uifigure` function do not have the option to specify `MenuBar` and `ToolBar` properties. The default menu and toolbar in figures created using the `figure` function is not supported.	Recreate the relevant behavior or design your own custom menu and toolbar by using the `uimenu` and `uitoolbar` functions.

Category	Not Supported	Suggested Actions
Container border properties	Certain options for configuring `Panel` and `ButtonGroup` objects are available only when the object is parented to a figure created using the `figure` function: • Specifying the `BorderType` property as `'etchedin'`, `'etchedout'`, `'beveledin'`, or `'beveledout'` • Specifying the `ShadowColor` property • Specifying the `TitlePosition` property as `'leftbottom'`, `'centerbottom'`, or `'rightbottom'`	Determine if this functionality is critical to your app before updating your app to use `uifigure`. There is no workaround in `uifigure`-based apps.
Modal dialog boxes	The `'modal'` option for `errordlg`, `warndlg`, `helpdlg`, `msgbox`, and `waitbar` has no effect when the dialog box is created for a `uifigure`-based app.	Replace calls to these functions with dialog box functions such as `uialert`, `uiconfirm`, and `uiprogressdlg`. These dialog boxes are created for use in `uifigure`-based apps and support modal options. For more information, see “Update Dialog Boxes” on page 14-18.

See Also

Related Examples

- “Update App Figure and Containers” on page 14-7
- “Update UIControl Objects and Callbacks” on page 14-11
- “Update Dialog Boxes” on page 14-18

Update App Figure and Containers

MATLAB provides two functions to create a figure window: `figure` and `uifigure`. The `uifigure` function is the recommended function to use when building new apps programmatically, and is the function that App Designer uses to create apps. The `figure` function will continue to be supported, but there are many new app building capabilities that can be used only with UI figures.

The first step in updating an app that uses the `figure` and `uicontrol` functions is to update your app figure and containers. You can do this by replacing calls to `figure` with `uifigure` and then updating your container layout. Most UI containers can be parented to a figure created using either the `figure` or the `uifigure` function, and so this step often requires minimal updates to your code.

Replace Calls to figure with uifigure

To transition your app to use modern app building functionality, first replace all calls to the `figure` function in your app code with calls to the `uifigure` function:

```
fig = uifigure;
```

Specify Target Object

After updating the figure creation function calls, if you plot data or create objects in your app without explicitly specifying the target object for the operation, running your code can create additional, unexpected figure windows. To address this behavior, further update your app code using one of these options:

- Specify the target or parent object in function calls — This is the best practice to avoid unexpected behavior. Most app building and graphics functions have an option for specifying the parent or target. For example, this code creates a panel in a UI figure by returning the `Figure` object as a variable and then providing that variable as the first input to the `uipanel` function.

  ```
  fig = uifigure;
  pnl = uipanel(fig);
  ```
- Set the `HandleVisibility` property value of the UI figure to `'callback'` — Use this option when your code that relies on objects in your app becoming the current object is invoked only from within callback functions in your app. When `HandleVisibility` is `'callback'`, the `Figure` object is visible only from within callbacks or functions invoked by callbacks, and not from within functions invoked from the Command Window.
- Set the `HandleVisibility` property value of the UI figure to `'on'` — Use this option to specify that the UI figure behavior is the same as the default behavior for figures created with the `figure` function. This option is not recommended because it can result in unexpected changes to the app UI.

Adjust Container Positions

Objects such as `Panel`, `TabGroup`, and `ButtonGroup` objects can be parented to figures created using either the `figure` or `uifigure` function. In general, these objects behave the same way in a `uifigure`-based app as they do in a `figure`-based app. However, some container objects have differences in default `Position` and `Units` properties.

If your app contains panels, tab groups, or button groups that are mispositioned after you transition to using the `uifigure` function, you have multiple options to update your code:

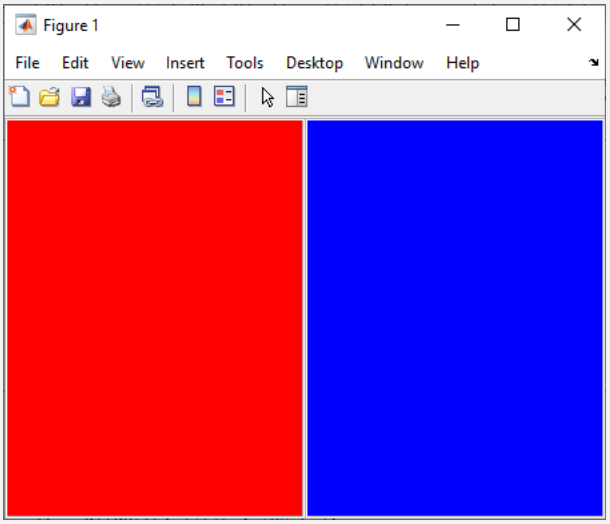
- “Use a Grid Layout Manager” on page 14-8 — Use this option if you want to refactor your app layout using modern layout tools. You can use a grid layout manager to align and specify the resize behavior of UI components by laying them out in a grid, which can greatly simplify your layout and resize code.
- “Specify Container Positions” on page 14-9 — Use this option if you want to quickly update your positioning code or if you want to continue to manage the layout of your app using the Position property and SizeChangedFcn callbacks.

Use a Grid Layout Manager

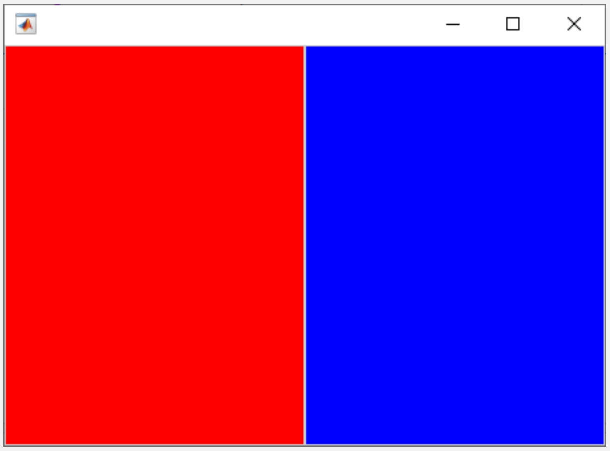
To manage your app layout and resize behavior relative to the size of the figure window, use a grid layout manager. Create a grid layout manager in your UI figure by using the uigridlayout function, and parent your app components and containers to the grid layout manager. For more information about using a grid layout manager to lay out your app, see “Lay Out Apps Programmatically” on page 10-2.

This table shows an example of a figure-based app with two panels laid out using the Position property, and the updated uifigure-based app laid out using a grid layout manager.

Code	App

Panels in a figure-based app, laid out using the Position property
`f = figure;`
`f.Position = [500 500 450 300];`
 	
`p1 = uipanel(f);`
`p1.Position = [0 0 0.5 1];`
`p1.BackgroundColor = "red";`
 	
`p2 = uipanel(f);`
`p2.Position = [0.5 0 0.5 1];`
`p2.BackgroundColor = "blue";`
	

Code	App

Panels in a `uifigure`-based app, laid out using a grid layout manager
`f = uifigure;`
`f.Position = [500 500 450 300];`
 	
`gl = uigridlayout(f,[1 2]);`
`gl.Padding = [0 0 0 0];`
`gl.ColumnSpacing = 0;`
 	
`p1 = uipanel(gl);`
`p1.Layout.Row = 1;`
`p1.Layout.Column = 1;`
`p1.BackgroundColor = "red";`
 	
`p2 = uipanel(gl);`
`p2.Layout.Row = 1;`
`p2.Layout.Column = 2;`
`p2.BackgroundColor = "blue";`
	

Specify Container Positions

Alternatively, you can continue to use the `Position` property to lay out your app. While `Panel`, `TabGroup`, and `ButtonGroup` objects that are parented to a figure created using the `figure` function use normalized units for their `Position` by default, these containers in a UI figure use pixel units by default instead. Pixel units are recommended for app building because most MATLAB app building functionality measures distances in pixels.

Follow these steps to update the property values of the `Panel`, `TabGroup`, and `ButtonGroup` objects in your app to use pixel units:

1. In your `figure`-based app, after laying out the object, set the value of its `Units` property to `"pixels"`, and then query the value of its `Position` property.

   For example, this code creates two panels laid out using normalized units, converts the units to pixels, and displays the corresponding pixel position values.

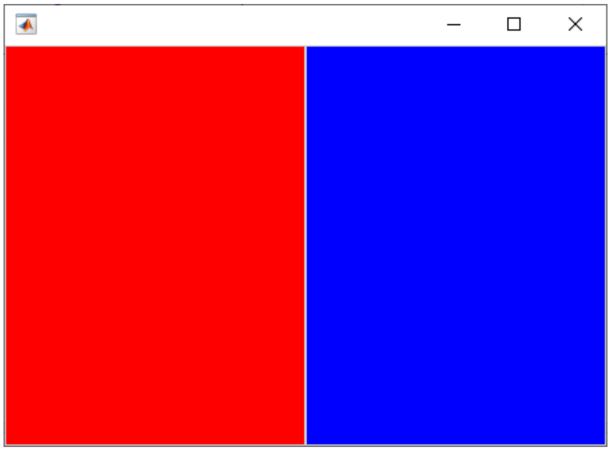
   ```
   f = figure;
   f.Position = [500 500 450 300];

   p1 = uipanel(f);
   p1.Position = [0 0 0.5 1];
   p1.BackgroundColor = "red";

   p2 = uipanel(f);
   p2.Position = [0.5 0 0.5 1];
   p2.BackgroundColor = "blue";

   p1.Units = "pixels";
   p2.Units = "pixels";

   p1PixelPosition = p1.Position
   p2PixelPosition = p2.Position
   ```

   ```
   p1PixelPosition =
   ```

```
     1     1   225   300


p2PixelPosition =

   226     1   225   300
```

2 In your `uifigure`-based app, set the `Position` property of each object to the equivalent pixel-based position.

```
f = uifigure;
f.Position = [500 500 450 300];

p1 = uipanel(f);
p1.Position = [1 1 225 300];
p1.BackgroundColor = "red";

p2 = uipanel(f);
p2.Position = [226 1 225 300];
p2.BackgroundColor = "blue";
```



See Also

Related Examples

- “Update figure-Based Apps to Use uifigure” on page 14-2
- “Update UIControl Objects and Callbacks” on page 14-11
- “Update Dialog Boxes” on page 14-18

Update UIControl Objects and Callbacks

In apps created using the `uifigure` function, add UI components to your app using component functions such as `uibutton` and `uidropdown`. Creating apps using the `figure` and `uicontrol` functions will continue to be supported. However, there are benefits to using UI components in a `uifigure`-based app over `UIControl` objects in a `figure`-based app. For example, these are some functionalities that exist only in `uifigure`-based apps:

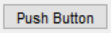
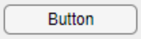
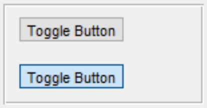
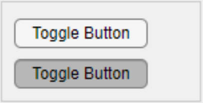
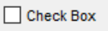
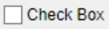
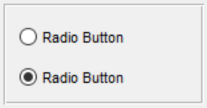
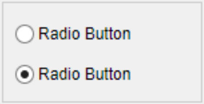
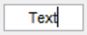
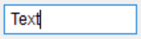
- New component types, such as trees, hyperlinks, and instrumentation components
- Layout tools to configure your app layout, such as grid layout managers
- Additional component customization options, such as component properties that control text alignment and component placeholder text

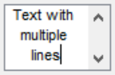
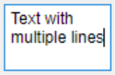
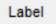
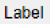
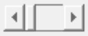
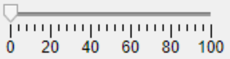
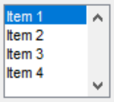
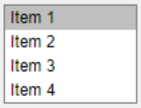
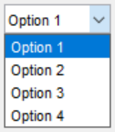
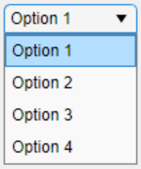
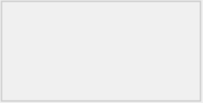
To take advantage of these benefits, transition your `figure`-based app to use the `uifigure` function. Then, follow these steps to replace `UIControl` objects in your app with UI components:

1. Replace calls to the `uicontrol` function with calls to the corresponding UI component function.
2. Update properties of the UI component.
3. Update callbacks of the UI component.

Replace uicontrol Function Calls

The `uicontrol` function has an argument for specifying the style of the control. Every `UIControl` style corresponds to a UI component object with similar functionality and appearance. In `uifigure`-based apps, replace calls to the `uicontrol` function with the corresponding UI component function. This table provides a list of the `UIControl` styles and the corresponding UI component function.

`UIControl` Objects		UI Component Objects	
Style	**Appearance**	**Function**	**Appearance**
`'pushbutton'`		`uibutton`	
`'togglebutton'`		• `uibutton` with `'state'` style for a single, independent state button • `uitogglebutton` for a group of linked toggle buttons	
`'checkbox'`		`uicheckbox`	
`'radiobutton'`		`uiradiobutton`	
`'edit'` (single line)		`uieditfield`	

UIControl Objects		**UI Component Objects**	
Style	**Appearance**	**Function**	**Appearance**
`'edit'` (multiple lines)		`uitextarea`	
`'text'`		`uilabel`	
`'slider'`		`uislider`	
`'listbox'`		`uilistbox`	
`'popupmenu'`		`uidropdown`	
`'frame'`		• `uipanel` • `uibuttongroup`	

Some UI components have slightly different configurations and behavior than their `UIControl` equivalent. In many cases, you can update your code to adjust for these differences using the following steps.

Slider Differences

The slider UI component created using `uislider` has a different appearance than the slider `UIControl` object.

If your app uses a `UIControl` slider to allow users to scroll in a container, consider removing your code that manages scrolling and using these alternatives instead:

- Set the `Scrollable` property of the container to `'on'` to enable scrolling.
- Use the `scroll` function to scroll within the container programmatically.

Text Input Differences

The `'edit'` style `UIControl` objects align text in the center by default, whereas `uieditfield` and `uitextarea` UI components align text on the left. You can specify the text alignment of these UI components by specifying the `HorizontalAlignment` property.

If your app uses an `'edit'` style `UIControl` object to allow users to input numeric values, you can instead create a numeric edit field using the `uieditfield` function by specifying the `style` argument as `"numeric"`:

```
fig = uifigure;
ef = uieditfield(fig,"numeric");
```

Parent Container Differences

Both the `uicontrol` function and UI component functions have an optional first input argument to specify the parent container. If you omit this argument in the `uicontrol` function, the function adds the control to the current figure. If you omit this argument in a UI component function such as `uibutton` or `uicheckbox`, the function creates a new UI figure and adds the component to that figure.

In `uifigure`-based apps, create the main app UI figure using the `uifigure` function and return the `Figure` object as a variable. Then, pass that variable as the first argument to the UI component functions.

```
fig = uifigure;
btn = uibutton(fig,"BackgroundColor","blue");
cbx = uicheckbox(fig,"Position",[220 100 84 22]);
```

For more information, see “Update App Figure and Containers” on page 14-7.

Update Component Properties

UI component objects and `UIControl` objects have many of the same properties. For example, both types of objects have `Position`, `BackgroundColor`, and `FontSize` properties. You can use the same code to set these properties for both `UIControl` objects and UI components.

However, if you set certain `UIControl` properties in your app, you might need to update the names or values of these properties when you transition to using UI components. This table lists some common properties of `UIControl` objects that differ from UI component properties and suggested actions to take if you set these properties in your code. If you encounter an error related to a property that is not listed in the table, see the properties page of the specific UI component to resolve the error. For a list of all UI components and links to their properties, see “App Building Components” on page 4-2.

`UIControl` Property	Description	Suggested Actions
`String`	The `String` property of a `UIControl` object specifies the display text for the component. Depending on the UI component, this property is replaced by `Text`, `Value`, or `Items`.	• Labels, buttons, and check boxes — Replace references to `String` with `Text`. • Edit fields and text areas — Replace references to `String` with `Value`. • Drop-down components and list boxes — Replace references to `String` with `Items`.

UIControl Property	Description	Suggested Actions
Units	The Units property of a UIControl object specifies the units of measurement for the object. UI component objects do not have a Units property. All UI components use pixel units to measure distances.	Update the Position property of your UI components to use pixel units. Alternatively, if you use normalized units to manage app resize behavior, instead update your app layout to use a grid layout manager. For more information, see “Manage App Resize Behavior Programmatically” on page 10-10.
Value	The Value property modifies the status of certain UIControl objects. For each of these UIControl styles, the equivalent UI component also has a Value property. However, the types of property values you specify might differ.	• State buttons, toggle buttons, radio buttons, and check boxes — Specify Value as 0 (unselected or raised) or 1 (selected or depressed). • Drop-down components and list boxes — Specify Value as an element of Items. • Sliders — No changes needed. The Value property for sliders has the same effect in UIControl objects and Slider UI components.
ForegroundColor	The ForegroundColor property of a UIControl object specifies the text color for the component. In UI components, this property is named FontColor.	Replace all references to ForegroundColor with FontColor.
Max and Min	The values of the Max and Min properties have different effects depending on the UIControl style. UI components have separate properties with more specific names and behavior.	• Sliders — Use the Limits property to set the maximum and minimum slider values. • Edit fields and text areas — Create an edit field for single-line text and a text area for multi-line text. • List boxes — Set the MultiSelect property to 'on' to allow users to select multiple items.

UIControl Property	Description	Suggested Actions
CData	The CData property specifies an icon or image associated with a UIControl object. UI components that support icons have an Icon property instead. In addition, there is an image UI component for displaying images in an app.	• Push buttons and toggle buttons — Specify Icon as a 3-D array of truecolor RGB values or a path to an image file. • Standalone images — Use the uiimage function.
Extent	The Extent property of the UIControl object stores the size of the object based on its text and font size. UI components have no equivalent property.	If your app uses the Extent property to specify a component size based on the text it contains, update your app layout to use a grid layout manager by using the uigridlayout function. Specify the column width of grid columns that contain components with text as 'fit', which scales the component size to fit the text it contains.
SliderStep	The SliderStep property controls the magnitude of the slider value change when a user clicks the arrow buttons. There is no equivalent functionality for a Slider object created using the uislider function.	Determine if this functionality is critical to your app before updating. There is no equivalent functionality in uifigure-based apps.

Update Callbacks

UIControl objects have a Callback property. The callback function assigned to this property executes in response to a user interaction, where the interaction depends on the style of the UIControl. For every UIControl style, the corresponding UI component has an equivalent callback property, but the property name is specific to the user interaction it corresponds to. To transition your app code, wherever you assign a callback function to a Callback property, update the property name to the equivalent callback property for the UI component. This table lists the callback property names for each component type.

UIControl Style	Callback User Interaction	Equivalent UI Component Callback
'pushbutton'	The user clicks the button.	ButtonPushedFcn
'togglebutton'	The user clicks the button.	ButtonPushedFcn
'checkbox'	The user sets or clears the check box.	SelectionChangedFcn
'radiobutton'	The user clicks the button.	SelectionChangedFcn of the parent ButtonGroup container

UIControl Style	Callback User Interaction	Equivalent UI Component Callback
'edit'	The user enters text in the edit field.	ValueChangedFcn
'slider'	The user changes the slider value.	ValueChangedFcn
'listbox'	The user selects an item.	ValueChangedFcn
'popupmenu'	The user selects an item.	ValueChangedFcn

For example, this code creates a button `UIControl` object that prints a statement to the MATLAB Command Window when the user pushes the button.

```
c = uicontrol;
c.Style = "pushbutton";
c.Callback = @(src,event)disp("Button pushed");
```

The behavior is equivalent to creating a `uibutton` component and setting the `ButtonPushedFcn` callback property:

```
fig = uifigure;
btn = uibutton(fig)
btn.ButtonPushedFcn = @(src,event)disp("Button pushed");
```

If your app uses a `KeyPressFcn` callback to respond while a user types in an `'edit'` style `UIControl` object, instead consider using the `ValueChangingFcn` callback when you update your `uicontrol` function to `uieditfield` or `uitextarea`. The `ValueChangingFcn` callback of an edit field or text area component executes repeatedly as the user types in the component.

```
fig = uifigure;
ef = uieditfield(fig);
ef.ValueChangingFcn = @(src,event)disp("Typing...");
```

Key Press and Button Down Callbacks

All `UIControl` objects have a `ButtonDownFcn` callback to respond when a user clicks on an object, and `KeyPressFcn` and `KeyReleaseFcn` callbacks to respond when a user presses a key when the object has focus. There is no equivalent callback associated with UI components. However, you can update your code to have the same behavior by specifying a `WindowButtonDownFcn`, `WindowKeyPressFcn`, or `WindowKeyReleaseFcn` callback on the UI figure that contains the component. You can then query the object that was last clicked by using the `CurrentObject` property.

UIControl Callback	UIFigure Callback	Example

ButtonDownFcn	WindowButtonDownFcn	`fig = uifigure;`
		`lb = uilistbox(fig);`
		`fig.WindowButtonDownFcn = {@processClic`
		
		`function processClick(src,event,lb)`
		`    if src.CurrentObject == lb`
		`        disp("List box clicked")`
		`    end`
		`end`

UIControl Callback	UIFigure Callback	Example

KeyPressFcn	WindowKeyPressFcn	fig = uifigure;
		lb = uilistbox(fig);
		fig.WindowKeyPressFcn = {@processKeyPre
		
		function processKeyPress(src,event,lb)
		    if src.CurrentObject == lb
		        disp("List box key pressed")
		    end
		end

KeyReleaseFcn	WindowKeyReleaseFcn	fig = uifigure;
		lb = uilistbox(fig);
		fig.WindowKeyReleaseFcn = {@processKeyR
		
		function processKeyRelease(src,event,lb
		    if src.CurrentObject == lb
		        disp("List box key released")
		    end
		end

See Also

Related Examples

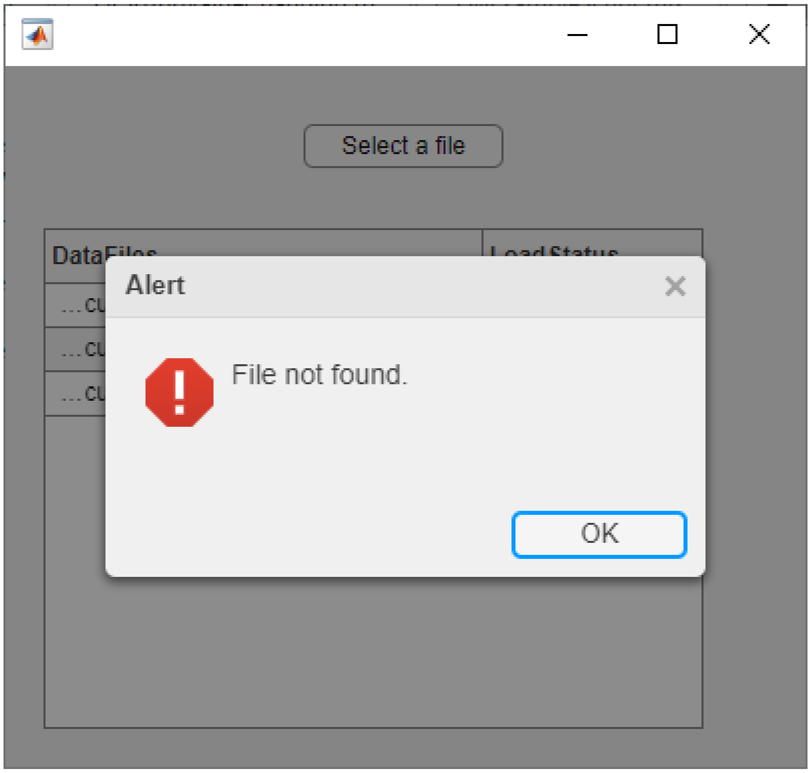
- “Update figure-Based Apps to Use uifigure” on page 14-2
- “Update App Figure and Containers” on page 14-7
- “Update Dialog Boxes” on page 14-18
- “Create and Run a Simple Programmatic App” on page 15-2

Update Dialog Boxes

Add dialog boxes to your `uifigure`-based app by using functions such as `uialert` and `uiconfirm`. These dialog box functions are specifically configured to be used in apps. Creating dialog boxes using functions such as `errordlg` and `questdlg` will continue to be supported. However, there are benefits to using dialog boxes specific to app building. These dialog boxes have additional customization options, including:

- The ability to specify a custom icon
- The ability to format text using HTML or LaTeX markup
- The ability to write a callback that executes when the dialog box is closed

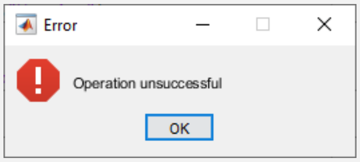
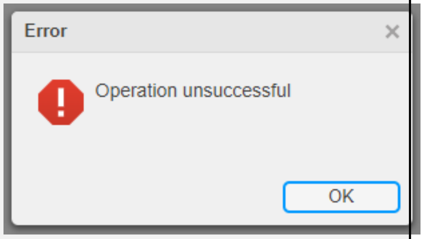
Also, these dialog boxes are displayed within the UI figure window that makes up your app.

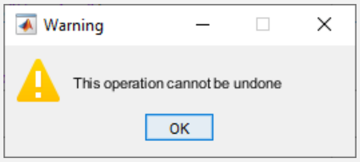
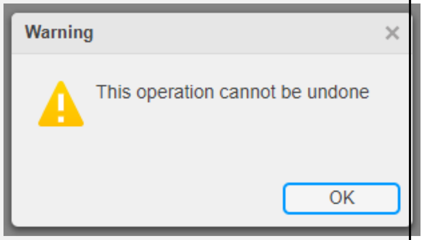


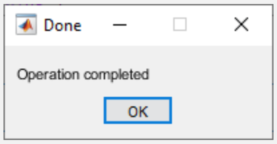
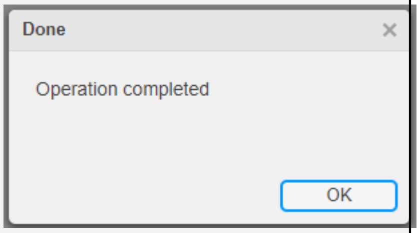
File not found.

To take advantage of these benefits, as you transition your `figure`-based app to use the `uifigure` function, update the functions you call to create dialog boxes for your app. This table lists the functions available for creating dialog boxes in `figure`-based apps and the corresponding functions configured for `uifigure`-based apps.

figure-Based Apps		uifigure-Based Apps	
Function	**Example**	**Function**	**Example**

`errordlg`	`errordlg("Operation unsuccessful","Error");`
	
	Error
	Operation unsuccessful
	
		`uialert`	`fig = uifigure;`
			`uialert(fig,"Operation unsuccessful","`
			
			Error
			Operation unsuccessful

`warndlg`	`warndlg("This operation cannot be undone","Warning");`
	
	Warning
	This operation cannot be undone
	
		`uialert`	`fig = uifigure;`
			`uialert(fig,"This operation cannot be`
			`    "Icon","warning")`
			
			Warning
			This operation cannot be undone

`msgbox`	`msgbox("Operation completed","Done","modal");`
	
	Done
	Operation completed
	
		`uialert`	`fig = uifigure;`
			`uialert(fig,"Operation completed","Don`
			`    "Icon","none")`
			
			Done
			Operation completed

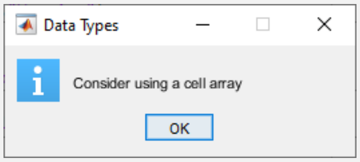
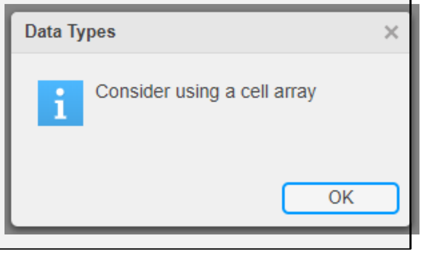
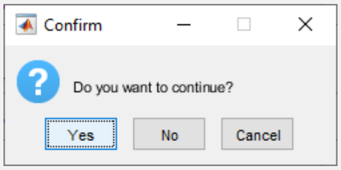
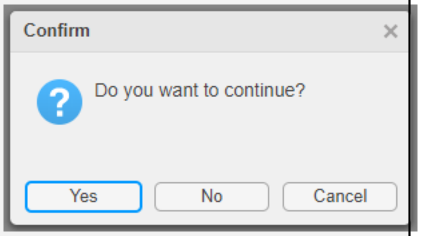
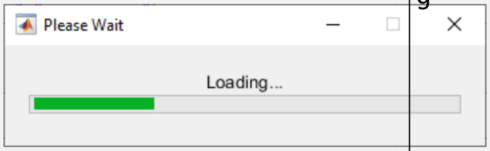
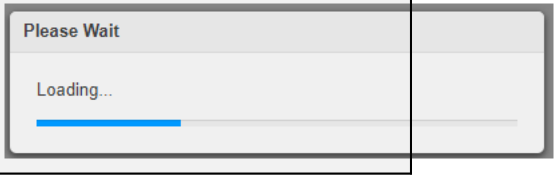
`helpdlg`	`helpdlg("Consider using a cell array","Data Types");`
	
	Data Types
	Consider using a cell array
	
		`uialert`	`fig = uifigure;`
			`uialert(fig,"Consider using a cell arr`
			`    "Icon","info")`
			
			Data Types
			Consider using a cell array

figure-Based Apps		uifigure-Based Apps	
Function	**Example**	**Function**	**Example**

`questdlg`	`questdlg("Do you want to continue?","Confirm");`
	
	
	Do you want to continue?
		`uiconfirm`	`fig = uifigure;`
			`uiconfirm(fig,"Do you want to continue`
			`    "Options",["Yes" "No" "Cancel"])`
			
			
			Do you want to continue?

`waitbar`	`waitbar(0.3,"Loading...","Name","Please Wait");`
	
	
	Loading...
		`uiprogressdlg`	`fig = uifigure;`
			`uiprogressdlg(fig,"Value",0.3, ...`
			`    "Message","Loading...", ...`
			`    "Title","Please Wait");`
			
			
			Loading...

See Also

Related Examples

- “Update figure-Based Apps to Use uifigure” on page 14-2
- “Update App Figure and Containers” on page 14-7
- “Update UIControl Objects and Callbacks” on page 14-11

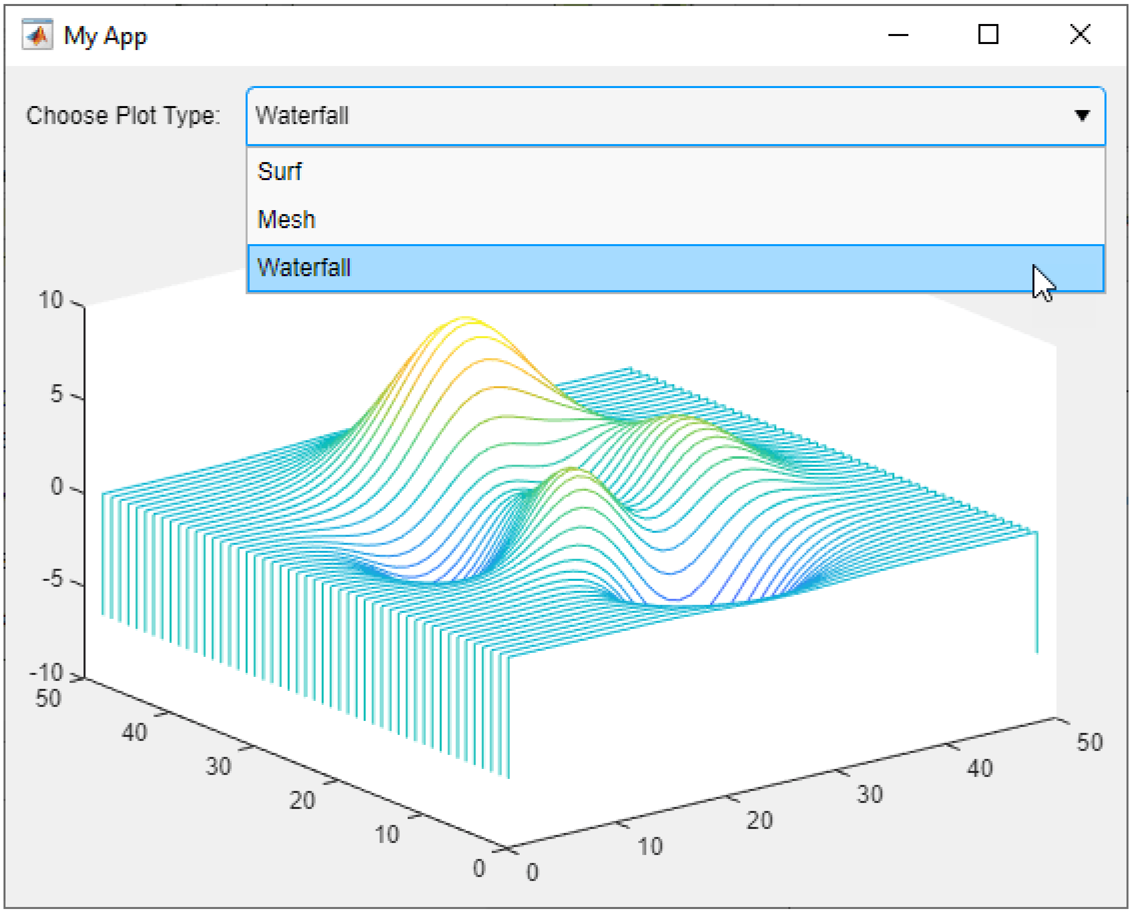
15

Examples of Programmatic Apps

Create and Run a Simple Programmatic App

This example shows how to create and run a programmatic app using MATLAB® functions. The example guides you through the process of building a runnable app in which users can interactively explore different types of plots. Build the app using these steps:

1. Design the app layout by creating the main figure window, laying out the UI components in it, and configuring the appearance of the components by setting properties.
2. Program the app to respond when a user interacts with it.
3. Run the app to verify that your app looks and behaves as expected.



Choose Plot Type: Waterfall

Define Main App Function

To create a programmatic app, write your app code in a function file. This allows users to run your app from the Command Window by entering the name of the function.

Create a new function named `simpleApp` and save it to a file named `simpleApp.m` in a folder that is on the MATLAB path. Provide context and instructions for using the app by adding help text to your function. Users can see this help text by entering `help simpleApp` in the Command Window.

```
function simpleApp
% SIMPLEAPP Interactively explore plotting functions
%   Choose the function used to plot the sample data to see the
%   differences between surface plots, mesh plots, and waterfall plots

end
```

Write all of your app code inside the simpleApp.m file. To view the full example code, see Run the App on page 15-5.

Create UI Figure Window

Every programmatic app requires a UI figure window to serve as the primary app container. This is the window that appears when a user runs your app, and it holds the UI components that make up the app. Create a UI figure window configured specifically for app building by calling the uifigure function. Return the resulting Figure object as a variable so that you can access the object later in your code. You can modify the size, appearance, and behavior of your figure window by setting figure properties using dot notation.

In this example, add this code to the simpleApp function to create a UI figure window and specify its title.

```
fig = uifigure;
fig.Name = "My App";
```

Manage App Layout

Manage the position and size of UI components in your figure window using a grid layout manager. This allows you to lay out your UI components in a grid by specifying a row and column for each component.

Add a grid layout manager to your app by using the uigridlayout function. Create the grid in the figure window by passing in fig as the first argument, and then specify the grid size. In this example, create a 2-by-2 grid by adding this code to the simpleApp function.

```
gl = uigridlayout(fig,[2 2]);
```

Control the size of each grid row and column by setting the RowHeight and ColumnWidth properties of the grid layout manager. In this example, ensure that the focal point of your app is the plotted data. Specify that the top row of the app is 30 pixels tall, and that the second row fills the rest of the figure window. Fit the width of the first column to the content it holds.

```
gl.RowHeight = {30,'1x'};
gl.ColumnWidth = {'fit','1x'};
```

For more information about how to lay out apps, see “Lay Out Apps Programmatically” on page 10-2.

Create and Position UI Components

Users interact with your app by interacting with different UI components, such as buttons, drop-downs, or edit fields. For a list of all available UI components, see “App Building Components” on page 4-2.

This example uses three different UI components:

- A label to provide instruction
- A drop-down to let users choose a plotting function

- A set of axes to plot the data on

Create a UI component and add it to the grid by calling the corresponding component creation function and specifying the grid layout manager as the first input argument. Store the components as variables to access them later in your code. To create and store these three components, add this code to the `simpleApp` function.

```
lbl  = uilabel(gl);
dd = uidropdown(gl);
ax = uiaxes(gl);
```

After you create the components for your app, position them in the correct rows and columns of the grid. To do this, set the `Layout` property of each component. Position the label in the upper-left corner of the grid and the drop-down in the upper-right corner. Make the `Axes` object span both columns in the second row by specifying `Layout.Column` as a two-element vector.

```
% Position label
lbl.Layout.Row = 1;
lbl.Layout.Column = 1;
% Position drop-down
dd.Layout.Row = 1;
dd.Layout.Column = 2;
% Position axes
ax.Layout.Row = 2;
ax.Layout.Column = [1 2];
```

Configure UI Component Appearance

Every UI component object has many properties that determine its appearance. To change a property, set it using dot notation. For a list of component properties, see the corresponding properties page. For example, DropDown Properties lists all the properties of the drop-down component.

Modify the label text to provide context for the drop-down options by setting the `Text` property.

```
lbl.Text = "Choose Plot Type:";
```

Specify the plotting functions that users can choose from in the drop-down by setting the `Items` property. Set the value of the drop-down that the user sees when they first run the app.

```
dd.Items = ["Surf","Mesh","Waterfall"];
dd.Value = "Surf";
```

Program App Behavior

Program your app to respond to user interactions by using callback functions. A callback function is a function that executes when the app user performs a specific interaction, such as selecting a drop-down item. Every UI component has multiple callback properties, each of which corresponds to a different user interaction. Write a callback function and assign it to an appropriate callback property to control the behavior of your app.

In this example, program your app to update the plot when a user selects a new drop-down item. In the `simpleApp.m` file, after the `simpleApp` function, define a callback function named `changePlotType`. MATLAB automatically passes two input arguments to every callback function when the callback is triggered. These input arguments are often named `src` and `event`. The first argument contains the component that triggered the callback, and the second argument contains information about the user interaction. Define `changePlotType` to accept `src` and `event` in addition to a third input argument that specifies the axes to plot on. In the callback function, access the new drop-down value using the event argument and then use this value to determine how to update the plot data. Call the appropriate plotting function and specify the input axes as the axes to plot on.

```
function changePlotType(src,event,ax)
type = event.Value;
switch type
    case "Surf"
        surf(ax,peaks);
    case "Mesh"
        mesh(ax,peaks);
    case "Waterfall"
        waterfall(ax,peaks);
end
end
```

To associate the changePlotType function with the drop-down component, in the simpleApp function, set the ValueChangedFcn property of the drop-down component to be a cell array. The first element of the cell array is a handle to the changePlotType callback function. The second element is the Axes object to plot the data on. When an app user selects a drop-down option, MATLAB calls the callback function and passes in the source, event, and axes arguments. The callback function then updates the plot in the app.

```
dd.ValueChangedFcn = {@changePlotType,ax};
```

For more information about writing callback functions, see “Create Callbacks for Apps Created Programmatically” on page 11-2.

Finally, to make sure the plotted data is consistent with the drop-down value even before changePlotType first executes, call the surf function.

```
surf(ax,peaks);
```

Run the App

After adding all of the app elements, your simpleApp function should look like this:

```
function simpleApp
% SIMPLEAPP Interactively explore plotting functions
%   Choose the function used to plot the sample data to see the
%   differences between surface plots, mesh plots, and waterfall plots

% Create figure window
fig = uifigure;
fig.Name = "My App";

% Manage app layout
gl = uigridlayout(fig,[2 2]);
gl.RowHeight = {30,'1x'};
gl.ColumnWidth = {'fit','1x'};

% Create UI components
lbl = uilabel(gl);
dd = uidropdown(gl);
ax = uiaxes(gl);
```

```
% Lay out UI components
% Position label
lbl.Layout.Row = 1;
lbl.Layout.Column = 1;
% Position drop-down
dd.Layout.Row = 1;
dd.Layout.Column = 2;
% Position axes
ax.Layout.Row = 2;
ax.Layout.Column = [1 2];

% Configure UI component appearance
lbl.Text = "Choose Plot Type:";
dd.Items = ["Surf" "Mesh" "Waterfall"];
dd.Value = "Surf";
surf(ax,peaks);

% Assign callback function to drop-down
dd.ValueChangedFcn = {@changePlotType,ax};
end

% Program app behavior
function changePlotType(src,event,ax)
type = event.Value;
switch type
    case "Surf"
        surf(ax,peaks);
    case "Mesh"
        mesh(ax,peaks);
    case "Waterfall"
        waterfall(ax,peaks);
end
end
```

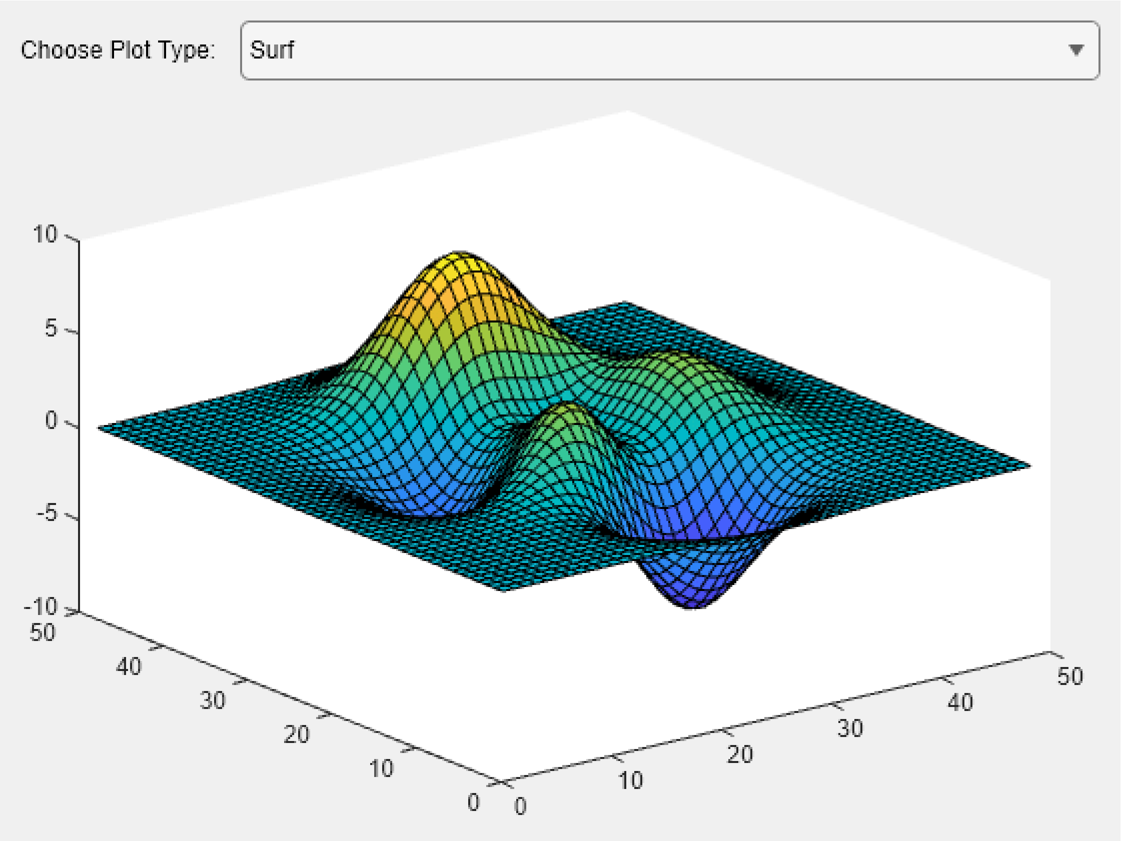
View the help text for your app.

```
help simpleApp
```

```
  SIMPLEAPP Interactively explore plotting functions
    Choose the function used to plot the sample data to see the
    differences between surface plots, mesh plots, and waterfall plots
```

Run the app by entering the app name in the Command Window. Update the plot by choosing a different plotting option from the drop-down.

```
simpleApp
```



Choose Plot Type: Surf

See Also

Related Examples

- “App Building Components” on page 4-2
- “Lay Out Apps Programmatically” on page 10-2
- “Create Callbacks for Apps Created Programmatically” on page 11-2
- “Create and Run a Simple App Using App Designer” on page 3-2

Programmatic App That Displays a Table

This example shows how to display a table in an app using the `uitable` function. It also shows how to modify the appearance of the table and how to restrict editing of the table in the running app.

Create a Table UI Component Within a Figure

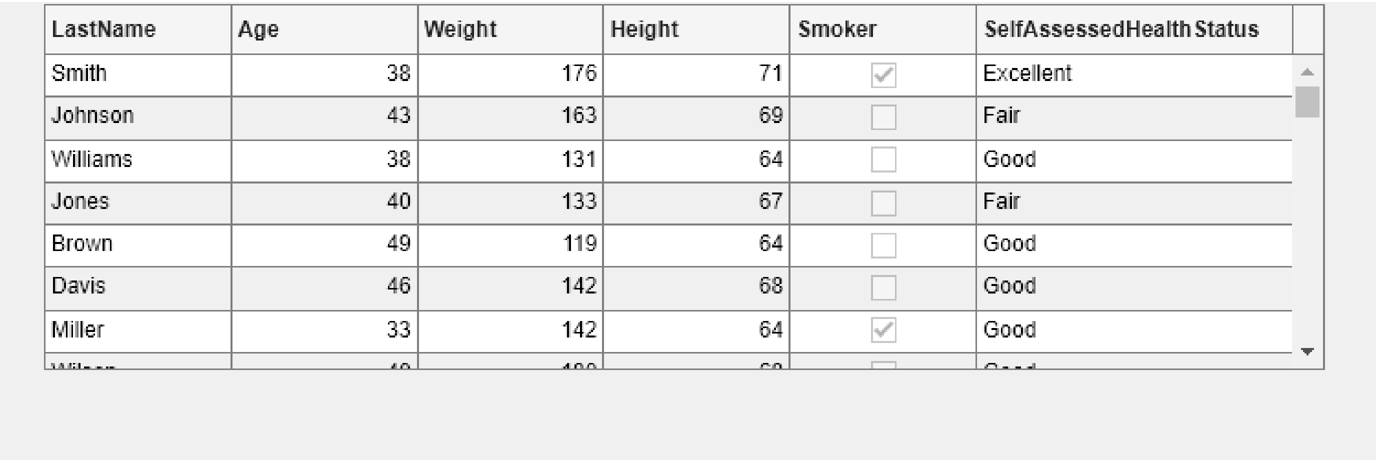
The `uitable` function creates an empty UI table in the figure.

```
fig = uifigure('Position',[100 100 752 250]);
uit = uitable('Parent',fig,'Position',[25 50 700 200]);
```

Create a Table Containing Mixed Data Types

Load sample patient data that contains mixed data types and store it in a table array. To have data appear as a drop-down list in the cells of the table component, convert a cell array variable to a categorical array. To display the data in the table UI component, specify the table array as the value of the `Data` property.

```
load patients
t = table(LastName,Age,Weight,Height,Smoker, ...
          SelfAssessedHealthStatus);
t.SelfAssessedHealthStatus = categorical(t.SelfAssessedHealthStatus, ...
          {'Poor','Fair','Good','Excellent'},'Ordinal',true);

uit.Data = t;
```

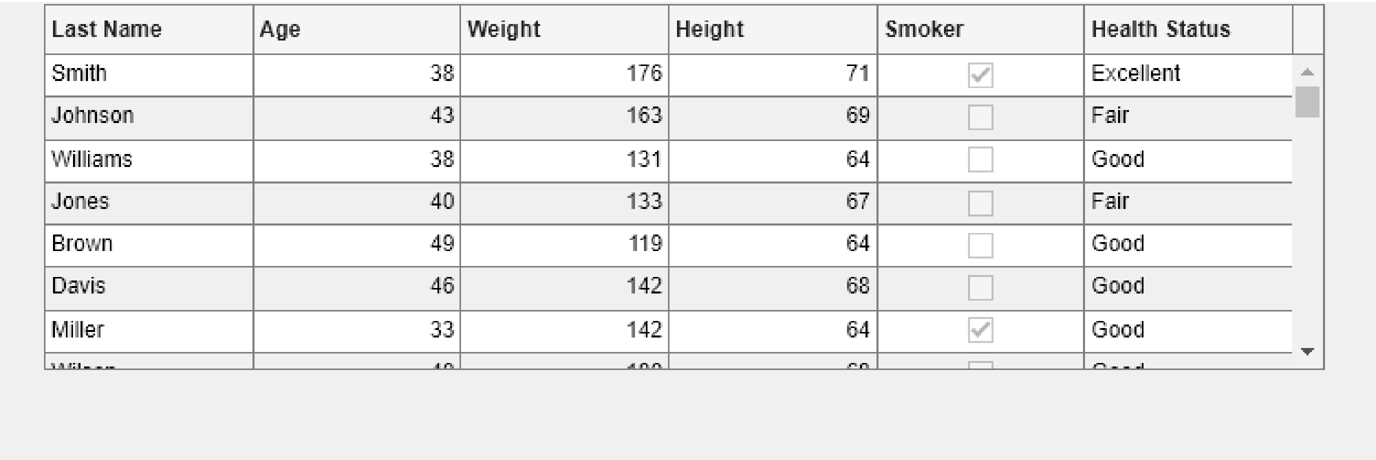


LastName	Age	Weight	Height	Smoker	SelfAssessedHealth Status
Smith	38	176	71	☑	Excellent
Johnson	43	163	69	□	Fair
Williams	38	131	64	□	Good
Jones	40	133	67	□	Fair
Brown	49	119	64	□	Good
Davis	46	142	68	□	Good
Miller	33	142	64	☑	Good

Customize the Display

You can customize the display of a UI table in several ways. Use the `ColumnName` property to add column headings.

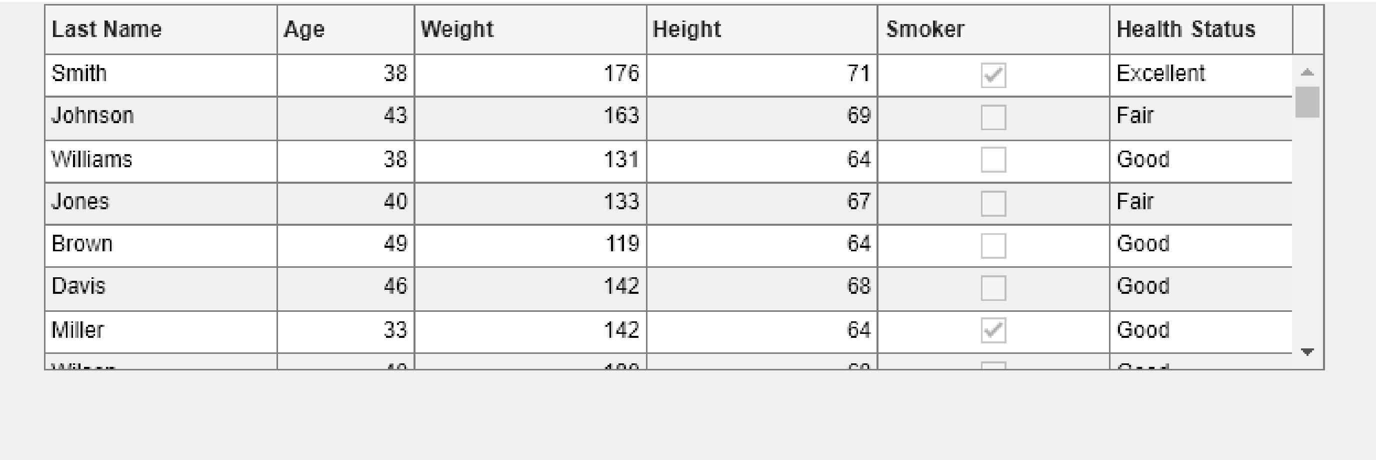
```
uit.ColumnName = {'Last Name','Age','Weight', ...
                  'Height','Smoker','Health Status'};
```



Last Name	Age	Weight	Height	Smoker	Health Status
Smith	38	176	71	☑	Excellent
Johnson	43	163	69	□	Fair
Williams	38	131	64	□	Good
Jones	40	133	67	□	Fair
Brown	49	119	64	□	Good
Davis	46	142	68	□	Good
Miller	33	142	64	☑	Good

To adjust the widths of the columns, specify the ColumnWidth property. The ColumnWidth property is a 1-by-N cell array, where N is the number of columns in the table. Set a specific column width, or use 'auto' to let MATLAB® set the width based on the contents.

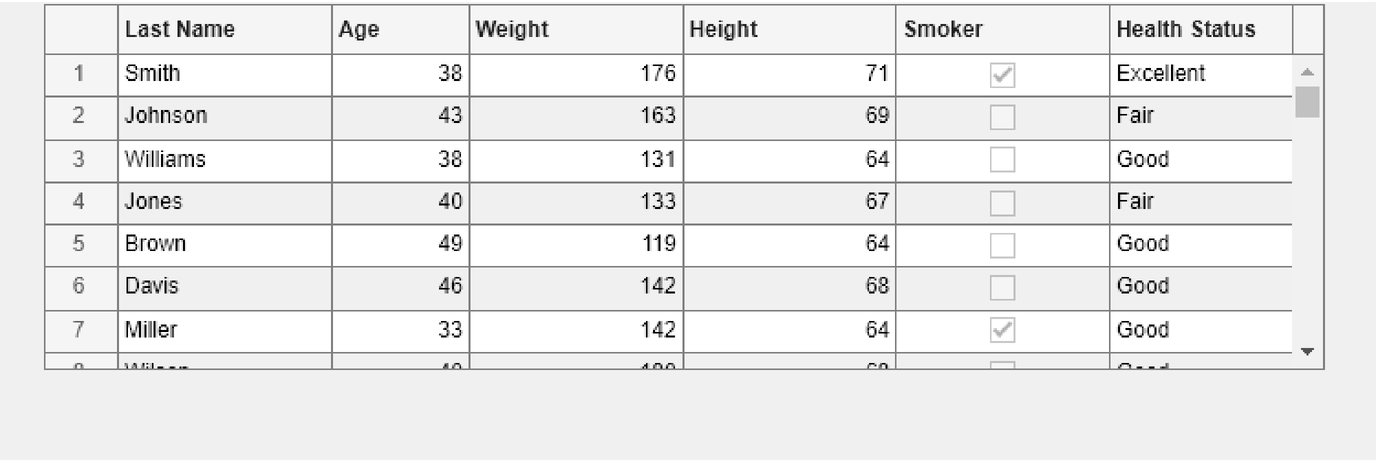
```
uit.ColumnWidth = {'auto',75,'auto','auto','auto',100};
```



Last Name	Age	Weight	Height	Smoker	Health Status
Smith	38	176	71	☑	Excellent
Johnson	43	163	69	□	Fair
Williams	38	131	64	□	Good
Jones	40	133	67	□	Fair
Brown	49	119	64	□	Good
Davis	46	142	68	□	Good
Miller	33	142	64	☑	Good

Add numbered row headings by setting the RowName property to 'numbered'.

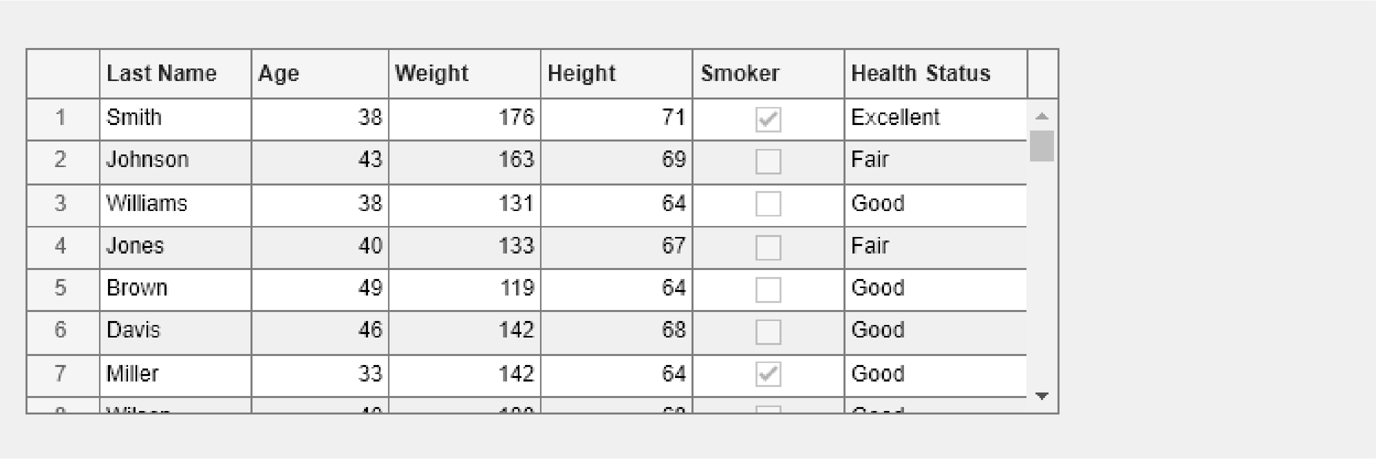
```
uit.RowName = 'numbered';
```



	Last Name	Age	Weight	Height	Smoker	Health Status
1	Smith	38	176	71	☑	Excellent
2	Johnson	43	163	69	□	Fair
3	Williams	38	131	64	□	Good
4	Jones	40	133	67	□	Fair
5	Brown	49	119	64	□	Good
6	Davis	46	142	68	□	Good
7	Miller	33	142	64	☑	Good

Reposition and resize the table using the `Position` property.

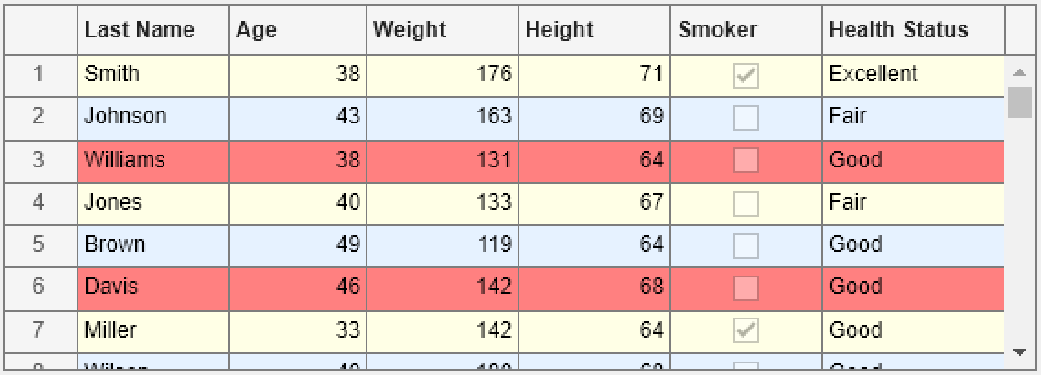
```
uit.Position = [15 25 565 200];
```



	Last Name	Age	Weight	Height	Smoker	Health Status
1	Smith	38	176	71	☑	Excellent
2	Johnson	43	163	69	□	Fair
3	Williams	38	131	64	□	Good
4	Jones	40	133	67	□	Fair
5	Brown	49	119	64	□	Good
6	Davis	46	142	68	□	Good
7	Miller	33	142	64	☑	Good

By default, table UI components use row striping and cycle through the specified background colors until the end of the table is reached. If you set the `RowStriping` property to `'off'`, the table UI component will just use the first color specified in the `BackgroundColor` property for all rows. Here, leave row striping `'on'` and set three different colors for the `BackgroundColor` property.

```
uit.BackgroundColor = [1 1 .9; .9 .95 1;1 .5 .5];
```

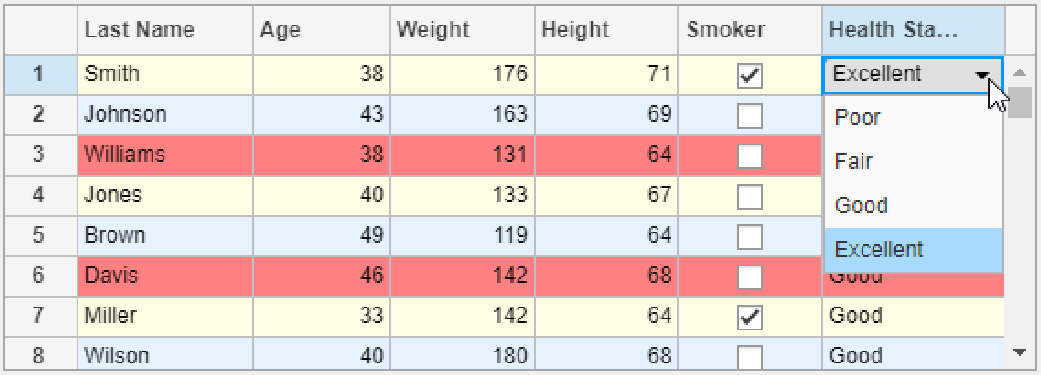


	Last Name	Age	Weight	Height	Smoker	Health Status
1	Smith	38	176	71	✓	Excellent
2	Johnson	43	163	69	□	Fair
3	Williams	38	131	64	□	Good
4	Jones	40	133	67	□	Fair
5	Brown	49	119	64	□	Good
6	Davis	46	142	68	□	Good
7	Miller	33	142	64	✓	Good

Enable Column Sorting and Restrict Editing of Cell Values

To restrict the user's ability to edit data in the table, set the `ColumnEditable` property. By default, data cannot be edited in the running app. Setting the `ColumnEditable` property to `true` for a column allows the user to edit data in that column.

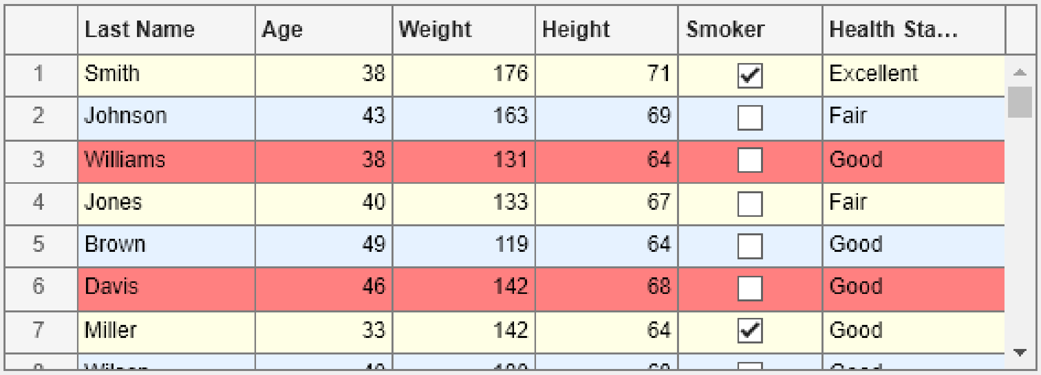
```
uit.ColumnEditable = [false true true true true true];
```



	Last Name	Age	Weight	Height	Smoker	Health Sta...
1	Smith	38	176	71	✓	Excellent
2	Johnson	43	163	69	□	
3	Williams	38	131	64	□	
4	Jones	40	133	67	□	
5	Brown	49	119	64	□	
6	Davis	46	142	68	□	Good
7	Miller	33	142	64	✓	Good
8	Wilson	40	180	68	□	Good

Make all the columns sortable by setting the `ColumnSortable` property to `true`. If a column is sortable, arrows appear in the header when you hover your mouse over it.

```
uit.ColumnSortable = true;
```



	Last Name	Age	Weight	Height	Smoker	Health Sta...
1	Smith	38	176	71	☑	Excellent
2	Johnson	43	163	69	□	Fair
3	Williams	38	131	64	□	Good
4	Jones	40	133	67	□	Fair
5	Brown	49	119	64	□	Good
6	Davis	46	142	68	□	Good
7	Miller	33	142	64	☑	Good

Create a Callback

To program the table to respond to user interaction, create a callback function. For example, you can specify a `SelectionChangedFcn` to execute commands when the app user selects a different row, column, or cell of the table.

Here, write a callback function to validate that the values in the `Age` column are between 0 and 120. Create a new function named `ageCheckCB` and save it to a file named `ageCheckCB` in a folder that is on the MATLAB path.

```
function ageCheckCB(src,event)
if (event.Indices(2) == 2 && ...                  % check if column 2
      (event.NewData < 0 || event.NewData > 120))
   tableData = src.Data;
   tableData{event.Indices(1),event.Indices(2)} = event.PreviousData;
   src.Data = tableData;                              % revert the data
   warning('Age must be between 0 and 120.')          % warn the user
end
```

Assign the `ageCheckCB` to the `CellEditCallback` property. This callback executes when the user changes a value in a cell. If the user enters a value that is outside the acceptable range, the callback function returns a warning and sets the cell value back to the previous value.

```
uit.CellEditCallback = @ageCheckCB;
```

For more information about writing callback functions, see “Create Callbacks for Apps Created Programmatically” on page 11-2.

Get All Table Properties

To see all the properties of the table, use the `get` command.

```
get(uit)
```

```
          BackgroundColor: [3x3 double]
             BeingDeleted: off
               BusyAction: 'queue'
            ButtonDownFcn: ''
         CellEditCallback: @ageCheckCB
    CellSelectionCallback: ''
                 Children: [0x0 handle]
               ClickedFcn: ''
           ColumnEditable: [0 1 1 1 1 1]
             ColumnFormat: {}
               ColumnName: {6x1 cell}
      ColumnRearrangeable: off
           ColumnSortable: 1
              ColumnWidth: {'auto'  [75]  'auto'  'auto'  'auto'  [100]}
              ContextMenu: [0x0 GraphicsPlaceholder]
                CreateFcn: ''
                     Data: [100x6 table]
                DeleteFcn: ''
              DisplayData: [100x6 table]
    DisplayDataChangedFcn: ''
         DoubleClickedFcn: ''
                   Enable: 'on'
                   Extent: [0 0 300 300]
                FontAngle: 'normal'
                 FontName: 'Helvetica'
                 FontSize: 12
                FontUnits: 'pixels'
               FontWeight: 'normal'
          ForegroundColor: [0 0 0]
         HandleVisibility: 'on'
            InnerPosition: [15 25 565 200]
            Interruptible: on
              KeyPressFcn: ''
            KeyReleaseFcn: ''
                   Layout: [0x0 matlab.ui.layout.LayoutOptions]
              Multiselect: on
            OuterPosition: [15 25 565 200]
                   Parent: [1x1 Figure]
                 Position: [15 25 565 200]
                  RowName: 'numbered'
              RowStriping: on
                Selection: []
      SelectionChangedFcn: ''
            SelectionType: 'cell'
      StyleConfigurations: [0x3 table]
                      Tag: ''
                  Tooltip: ''
                     Type: 'uitable'
                    Units: 'pixels'
                 UserData: []
                  Visible: on
```

See Also

Functions
`uitable` | `uifigure`

Properties
Table Properties

Related Examples

- "Style Cells in a Table UI Component" on page 15-15

- “Display Tabular Data in Apps” on page 4-15
- “Create App with a Table That Can Be Sorted and Edited Interactively” on page 7-9

Style Cells in a Table UI Component

When you display table data in an app, you can style individual cells, rows, and columns of the table UI component by using the `uistyle` and `addStyle` functions. Use styles to modify background colors, display icons, format equations, and provide clickable links in table cells. This example shows how you can use styles to display sample tsunami data in a table.

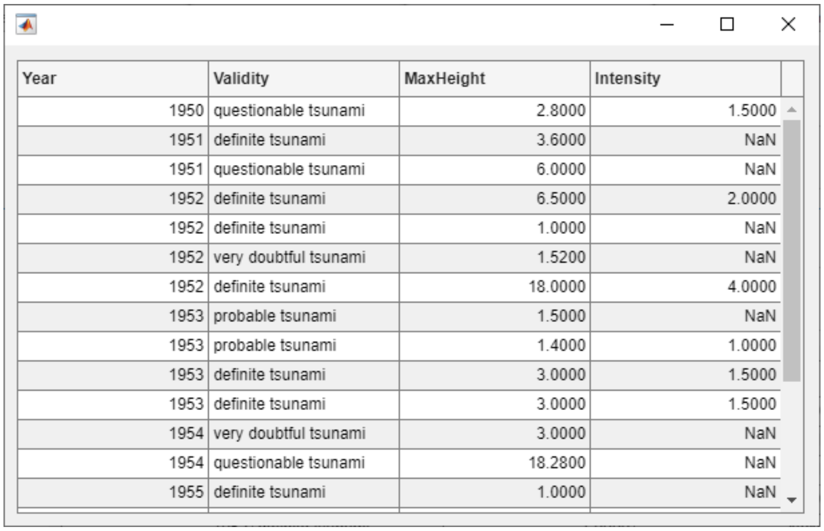
Create Table UI Component

Read in tsunami data from a file, and then extract a subset of the data to display.

```
T = readtable("tsunamis.xlsx");
T = T(1:20,["Year" "Validity" "MaxHeight" "Intensity"]);
```

Display the data in a table UI component.

```
fig = uifigure;
tbl = uitable(fig,"Position",[10 10 540 400],"Data",T);
```



Year	Validity	MaxHeight	Intensity
1950	questionable tsunami	2.8000	1.5000
1951	definite tsunami	3.6000	NaN
1951	questionable tsunami	6.0000	NaN
1952	definite tsunami	6.5000	2.0000
1952	definite tsunami	1.0000	NaN
1952	very doubtful tsunami	1.5200	NaN
1952	definite tsunami	18.0000	4.0000
1953	probable tsunami	1.5000	NaN
1953	probable tsunami	1.4000	1.0000
1953	definite tsunami	3.0000	1.5000
1953	definite tsunami	3.0000	1.5000
1954	very doubtful tsunami	3.0000	NaN
1954	questionable tsunami	18.2800	NaN
1955	definite tsunami	1.0000	NaN

Modify Background Color of Table Rows

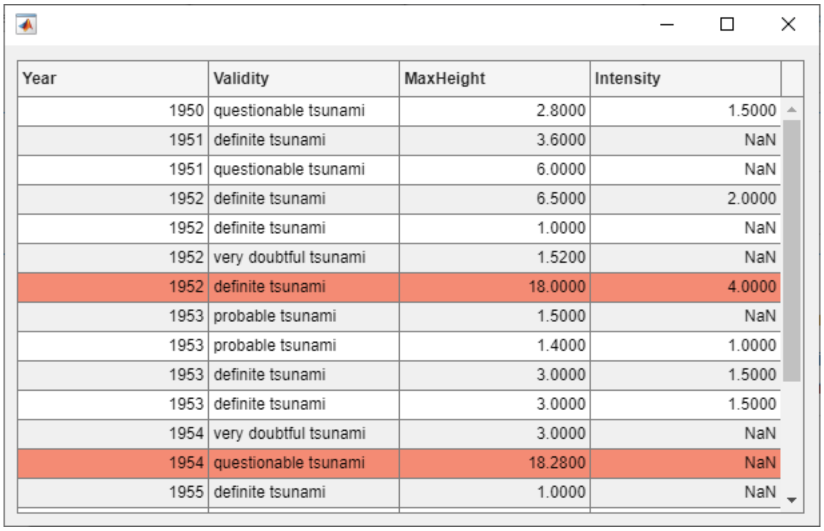
Draw attention to the rows of the table that represent the largest tsunamis by setting the background color of those rows to red.

First, find the rows that represent tsunamis with a maximum height greater than 10 meters.

```
rows = find(tbl.Data.MaxHeight > 10);
```

Then, create a style with a red background color and add the style to those rows.

```
s1 = uistyle("BackgroundColor","#F48B74");
addStyle(tbl,s1,"row",rows)
```



Year	Validity	MaxHeight	Intensity
1950	questionable tsunami	2.8000	1.5000
1951	definite tsunami	3.6000	NaN
1951	questionable tsunami	6.0000	NaN
1952	definite tsunami	6.5000	2.0000
1952	definite tsunami	1.0000	NaN
1952	very doubtful tsunami	1.5200	NaN
1952	definite tsunami	18.0000	4.0000
1953	probable tsunami	1.5000	NaN
1953	probable tsunami	1.4000	1.0000
1953	definite tsunami	3.0000	1.5000
1953	definite tsunami	3.0000	1.5000
1954	very doubtful tsunami	3.0000	NaN
1954	questionable tsunami	18.2800	NaN
1955	definite tsunami	1.0000	NaN

Display Icons in Table Cells

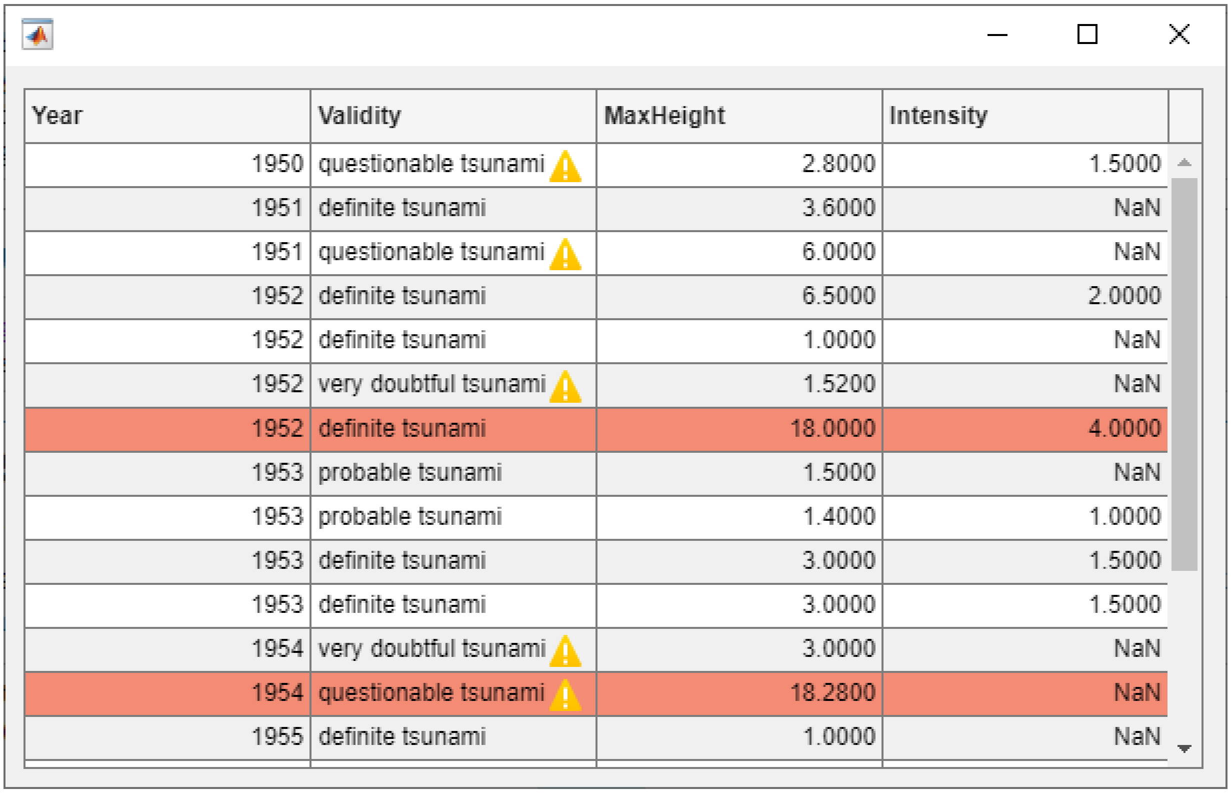
Next, add a warning icon to the table cells that specify that the tsunami validity is questionable or very doubtful.

Find the rows with questionable or doubtful tsunamis. Because all the cells that specify validity are in the second column of the table, construct an array containing the row and column indices of the cells by horizontally concatenating a vector that contains only values of 2.

```
warningRows = find(strcmp(tbl.Data.Validity,'questionable tsunami') | ...
    strcmp(tbl.Data.Validity,'very doubtful tsunami'));
warningColumns = repmat(2,size(warningRows));
cells = [warningRows warningColumns];
```

Finally, style the cells with a warning icon to the right of the text.

```
s2 = uistyle("Icon","warning","IconAlignment","right");
addStyle(tbl,s2,"cell",cells)
```



Year	Validity	MaxHeight	Intensity
1950	questionable tsunami ⚠	2.8000	1.5000
1951	definite tsunami	3.6000	NaN
1951	questionable tsunami ⚠	6.0000	NaN
1952	definite tsunami	6.5000	2.0000
1952	definite tsunami	1.0000	NaN
1952	very doubtful tsunami ⚠	1.5200	NaN
1952	definite tsunami	18.0000	4.0000
1953	probable tsunami	1.5000	NaN
1953	probable tsunami	1.4000	1.0000
1953	definite tsunami	3.0000	1.5000
1953	definite tsunami	3.0000	1.5000
1954	very doubtful tsunami ⚠	3.0000	NaN
1954	questionable tsunami ⚠	18.2800	NaN
1955	definite tsunami	1.0000	NaN

Format Equations and Symbols in Table Column

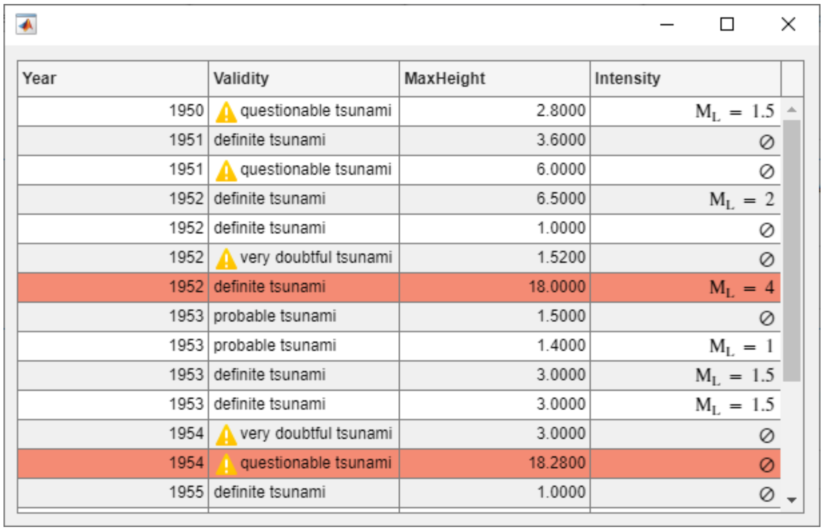
Add TeX markup to the `Intensity` column of the table.

First, identify the rows with no intensity data. Then, convert the values in the `Intensity` column to strings, and specify that the cells with missing data display the ⊘ symbol. For the cells with data, prepend the string `"M_L = "` to the data to indicate that the value gives the Richter magnitude.

```
nanData = isnan(tbl.Data.Intensity);
tbl.Data.Intensity = string(tbl.Data.Intensity);
tbl.Data.Intensity(nanData) = "\oslash";
tbl.Data.Intensity(~nanData) = "M_L = " + tbl.Data.Intensity(~nanData);
```

Style the cells in the column to use TeX markup by setting the `Interpreter` property to `"tex"`. Additionally, align text on the right side of the cells in the column by setting the `HorizontalAlignment` property to `"right"`.

```
s3 = uistyle("Interpreter","tex","HorizontalAlignment","right");
addStyle(tbl,s3,"column","Intensity")
```



Year	Validity	MaxHeight	Intensity
1950	⚠ questionable tsunami	2.8000	$M_L = 1.5$
1951	definite tsunami	3.6000	⊘
1951	⚠ questionable tsunami	6.0000	⊘
1952	definite tsunami	6.5000	$M_L = 2$
1952	definite tsunami	1.0000	⊘
1952	⚠ very doubtful tsunami	1.5200	⊘
1952	definite tsunami	18.0000	$M_L = 4$
1953	probable tsunami	1.5000	⊘
1953	probable tsunami	1.4000	$M_L = 1$
1953	definite tsunami	3.0000	$M_L = 1.5$
1953	definite tsunami	3.0000	$M_L = 1.5$
1954	⚠ very doubtful tsunami	3.0000	⊘
1954	⚠ questionable tsunami	18.2800	⊘
1955	definite tsunami	1.0000	⊘

Remove Style

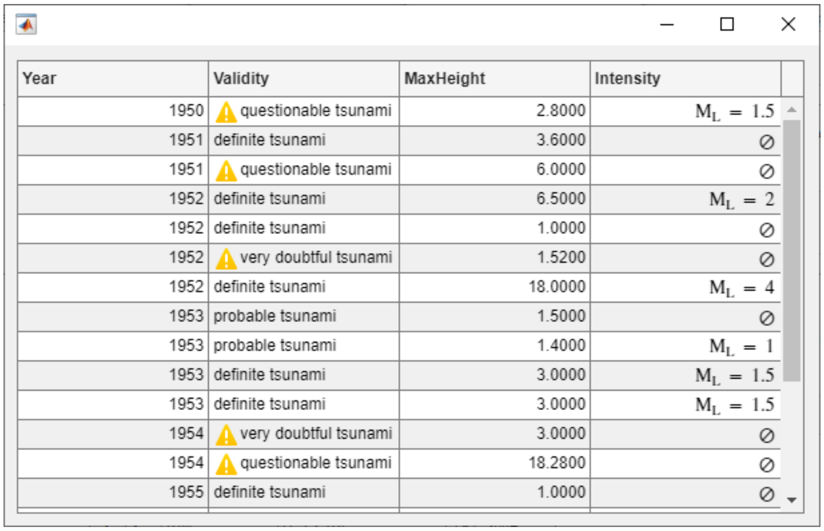
Inspect the styles on the table by querying the `StyleConfigurations` property of the table UI component. The styles are listed in the order in which you applied them to the table.

```
tbl.StyleConfigurations
```

```
ans =

  3×3 table

           Target      TargetIndex               Style
           ______    _____________    ________________________

    1      row       {[  7 13 16]}    1×1 matlab.ui.style.Style
    2      cell      {6×2 double }    1×1 matlab.ui.style.Style
    3      column    {'Intensity'}    1×1 matlab.ui.style.Style
```

Remove the first style from the table.

```
removeStyle(tbl,1)
```



Year	Validity	MaxHeight	Intensity
1950	⚠ questionable tsunami	2.8000	$M_L = 1.5$
1951	definite tsunami	3.6000	⊘
1951	⚠ questionable tsunami	6.0000	⊘
1952	definite tsunami	6.5000	$M_L = 2$
1952	definite tsunami	1.0000	⊘
1952	⚠ very doubtful tsunami	1.5200	⊘
1952	definite tsunami	18.0000	$M_L = 4$
1953	probable tsunami	1.5000	⊘
1953	probable tsunami	1.4000	$M_L = 1$
1953	definite tsunami	3.0000	$M_L = 1.5$
1953	definite tsunami	3.0000	$M_L = 1.5$
1954	⚠ very doubtful tsunami	3.0000	⊘
1954	⚠ questionable tsunami	18.2800	⊘
1955	definite tsunami	1.0000	⊘

See Also

Functions
`uitable` | `uistyle` | `addStyle` | `removeStyle` | `uifigure`

Properties
Table Properties

Related Examples

- “Programmatic App That Displays a Table” on page 15-8
- “Display Tabular Data in Apps” on page 4-15
- “Create App with a Table That Can Be Sorted and Edited Interactively” on page 7-9

Live Editor Task Development

Live Editor Task Development Overview

Live Editor tasks are simple point-and-click interfaces that can be embedded into a live script to perform a specific set of operations. You can use tasks to explore parameters and automatically generate code in a live script. Tasks are useful because they can help reduce development time, errors, and time spent plotting.

MATLAB provides a set of Live Editor tasks for use in live scripts. You also can create your own Live Editor task by defining a subclass of the `matlab.task.LiveTask` base class. Then, to make the task available in the Live Editor, configure the task using the Task Metadata dialog box.

Define Live Editor Task Subclass

To create custom Live Editor task, first, define a subclass of the `matlab.task.LiveTask` base class by following these steps:

1. Create the Live Editor task subclass.
2. Define public and private properties.
3. Implement the `get.Summary`, `get.State`, and `set.State` methods.
4. Implement the `setup`, `generateCode`, and `reset` methods.
5. Implement the `postExecutionUpdate` method (optional).
6. Emit a `StateChanged` event (optional).
7. Set the `AutoRun` property (optional).

Create Live Editor Task Subclass

To create a subclass of the `matlab.task.LiveTask` base class, create a class definition file using this code. Replace the class name in the code with the name of your Live Editor task class. Then, save the file as a `.m` file with the same name as the class.

```
classdef TaskClassName < matlab.task.LiveTask
    properties(Access = private,Transient)

    end
    properties(Dependent)
        State
        Summary
    end
    methods(Access = protected)
        function setup(task)

        end
    end
    methods
        function [code,outputs] = generateCode(task)
            code = "";
            outputs = {};
        end

        function summary = get.Summary(task)
            summary = "Task summary";
        end

        function state = get.State(task)
            state = struct;
        end

        function set.State(task,state)

        end

        function reset(task)
        end
    end
end
```

For example, to create a Live Editor task that normalizes vector data, create the file `NormalizeVectorData.m` and add the preceding code to the file. Then, rename the class to `NormalizeVectorData` by changing the first line of the file.

```
classdef NormalizeVectorData < matlab.task.LiveTask
```

Define Public and Private Properties

Define the properties for your class. In the private properties block, define properties to store the implementation details of your class that you want to hide. These properties store the underlying graphics and UI objects that make up your task, in addition to any calculated values that you want to store. Eventually, your class will use the data in the public properties to configure the underlying objects in the private properties. Set the `Transient` attribute for the private block to avoid storing redundant information if an instance of the task is saved.

In the public properties block, define the properties that the base class needs access to, including the two required public properties `State` and `Summary`. The `State` property stores the current state of the task, and the `Summary` property stores a dynamic summary of what the task does. Set the `Dependent` attribute for the public block to avoid access issues with the get and set methods for the properties.

For example, define the public and private properties for the `NormalizeVectorData` class.

```
properties(Access = private,Transient)
    InputDataDropDown              matlab.ui.control.DropDown
    MethodDropDown                 matlab.ui.control.DropDown
    ZscoreGrid                     matlab.ui.container.GridLayout
    ZscoreDropDown                 matlab.ui.control.DropDown
end

properties(Dependent)
    State
    Summary
end
```

Implement get.Summary Method

Define the `get.Summary` method for your class to dynamically generate the description of what the task does. The `get.Summary` method executes when the value of the `Summary` property is requested. The returned summary displays at the top of the task and remains visible when the task is collapsed.

Define the method in a method block with no arguments so that it is called when getting the value of the `Summary` property. The `get.Summary` method returns the generated description as a character array.

For example, implement the `get.Summary` method for the `NormalizeVectorData` class. Use the selected normalizing method in the `MethodDropDown` list and the selected input data in the `InputDataDropDown` list to generate a dynamic summary based on the current selection. If no input data is selected, set the summary to match the default description defined in the `liveTasks.json` file. To display variable or function names in monospaced font, surround them with backticks (``).

```
function summary = get.Summary(task)
    if isequal(task.InputDataDropDown.Value,"select variable")
        summary = "Normalize vector data";
    else
        switch task.MethodDropDown.Value
            case "zscore"
                methodString = " using z-score";
            case "norm"
                methodString = " using 2-norm";
            case "scale"
                methodString = " using scaling by standard deviation";
        end
        summary = "Normalized vector `" + task.InputDataDropDown.Value + ...
            "`" + methodString;
    end
end
```

Implement get.State and set.State Methods

Define the `get.State` and `set.State` methods for your class to get and set the current state of the UI objects in the task. The Live Editor uses these methods to restore a task to a specified state during copy, paste, undo, and redo operations, as well as when the live script containing the task is closed and reopened. The current state of the task is stored in a struct. When the live script is closed, the Live Editor uses the `jsonencode` function to convert the struct returned by `get.State` to JSON format and saves the encoded state with the live script. When the live script is reopened, the Live Editor converts the encoded state back to a struct, which is then used to set the current state of the task using `set.State`. Refer to the `jsonencode` function for more information about the data types it supports.

Define the methods in the same method block as the `get.Summary` method so that they are called when setting or getting the value of the `State` property. The `get.State` method returns a struct containing the current value of each UI object in the task. The `set.State` method takes a struct previously returned by the `get.State` method and uses the struct to set the value of each UI object in the task.

For example, implement the `get.State` and `set.State` methods for the `NormalizeVectorData` class. In the `get.State` method, create and return a struct with the value of each `DropDown` object in the task.

```
function state = get.State(task)
    state = struct;
    state.InputDataDropDownValue = task.InputDataDropDown.Value;
    state.MethodDropDownValue = task.MethodDropDown.Value;
    state.ZscoreDropDownValue = task.ZscoreDropDown.Value;
end
```

In the `set.State` method, use the values stored in the struct that is passed in to set the value of each `DropDown` object in the task.

```
function set.State(task,state)
    value = state.InputDataDropDownValue;
    if ~ismember(value, task.InputDataDropDown.ItemsData)
        task.InputDataDropDown.Items = [task.InputDataDropDown.Items {value}];
        task.InputDataDropDown.ItemsData = [task.InputDataDropDown.ItemsData {value}];
    end
    task.InputDataDropDown.Value = value;
    task.MethodDropDown.Value = state.MethodDropDownValue;
    task.ZscoreDropDown.Value = state.ZscoreDropDownValue;
    updateComponents(task);
end
```

Implement setup Method

Define the `setup` method for your class. The `setup` method sets the initial state of the task and executes once when MATLAB constructs the task. Define the `setup` method in a protected block so that only your class can execute it.

Use the `setup` method to:

- Create, layout, and configure the graphics and UI objects that make up the task.

- Program the behavior of objects within the task.
- Set the default values for the objects.

Note All graphics and UI objects for the task must be added to the task's grid layout manager, LayoutManager. If an object is added to the task directly, MATLAB throws an error.

For example, implement the setup method for the NormalizeVectorData class. In the setup method, call the createComponents, setComponentsToDefault, and updateComponents helper functions to create and arrange the components in the task, set all the components to their default values, and update the components.

```
function setup(task)
    createComponents(task);
    setComponentsToDefault(task);
    updateComponents(task);
end
```

Create a private methods block and define the createComponents, setComponentsToDefault, updateComponents, and populateWSDropdownItems helper functions.

In the createComponents function, call the uilabel, uigridlayout, and uidropdown functions to create and arrange Label, GridLayout, and DropDown objects, specifying the task's LayoutManager as the parent. Store those objects in the corresponding private properties. Specify the updateComponents method as the ValueChangedFcn callback that is called when the value of a drop-down list is changed. Specify the populateWSDropdownItems method as the DropDownOpeningFcn callback that is called when a drop-down list is opened.

```
methods(Access = private)
    function createComponents(task)
        g = uigridlayout(task.LayoutManager,[1,1]);
        g.RowHeight = {'fit' 'fit' 'fit' 'fit'};
        g.ColumnWidth = {'fit'};

        % Row 1: Select data section label
        uilabel(g,"Text","Select data","FontWeight","bold");

        % Row 2: Select data section components
        inputgrid = uigridlayout(g,"RowHeight",{'fit'},"ColumnWidth", ...
            {'fit','fit'},"Padding",0);
        uilabel(inputgrid,"Text","Input data");
        task.InputDataDropDown = uidropdown(inputgrid, ...
            "ValueChangedFcn",@task.updateComponents, ...
            "DropDownOpeningFcn",@task.populateWSDropdownItems);
        task.populateWSDropdownItems(task.InputDataDropDown);

        % Row 3: Specify method section label
        uilabel(g,"Text","Specify method","FontWeight","bold");

        % Row 4: Method section components
        methodgrid = uigridlayout(g,"RowHeight",{'fit'}, ...
            "ColumnWidth",{'fit','fit','fit'},"Padding",0);
        uilabel(methodgrid,"Text","Normalization method");
        task.MethodDropDown = uidropdown(methodgrid,"ValueChangedFcn", ...
            @task.updateComponents);
        task.MethodDropDown.Items = ["Z-score" "2-Norm" ...
            "Scale by standard deviation"];
        task.MethodDropDown.ItemsData = {'zscore' 'norm' 'scale'};

        % Subgrid 1 in method section
        task.ZscoreGrid = uigridlayout(methodgrid,"RowHeight",{'fit'}, ...
            "ColumnWidth",{'fit','fit'},"Padding",0);
        uilabel(task.ZscoreGrid,"Text","Deviation type");
        task.ZscoreDropDown = uidropdown(task.ZscoreGrid,"ValueChangedFcn", ...
            @task.updateComponents,"Items",{'Standard' 'Median absolute'}, ...
            "ItemsData",{'std' 'robust'},"Tooltip", ...
            "Center data to 0 and scale to deviation 1");
    end
    function setComponentsToDefault(task)
        task.MethodDropDown.Value = "zscore";
        task.ZscoreDropDown.Value = "std";
    end

    function updateComponents(task,source,~)
        hasData = ~isequal(task.InputDataDropDown.Value,"select variable");

        task.MethodDropDown.Enable = hasData;
        task.ZscoreDropDown.Enable = hasData;

        % Show only relevant subgrids
        task.ZscoreGrid.Visible = isequal(task.MethodDropDown.Value,"zscore");

        % Trigger the Live Editor to update the generated script
        notify(task,"StateChanged");
    end

    function populateWSDropdownItems(~,src,~)
        workspaceVariables = evalin("base","who");
        src.Items = ["select variable"; workspaceVariables];
        src.ItemsData = ["select variable"; workspaceVariables];
    end
end
```

Implement generateCode Method

Define the `generateCode` method for your class to generate the MATLAB commands and output for the task. This method executes when the task state changes, for example, when a user modifies a task parameter. The generated code displays in the code section of the task. When the live script section containing the task runs, the Live Editor uses the generated code to run the task. Define the `generateCode` method in the same method block as the `get.Summary`, `get.State`, and `set.State` methods.

The `generateCode` method returns two output arguments, `code` and `outputs`. `code` is a character array or string array containing the generated code for the task. `outputs` is a cell array containing the output variables produced by the code. If the task does not generate output, return `outputs` as an empty cell array.

For example, implement the `generateCode` method for the `NormalizeVectorData` class:

- Start the generated code with a comment that describes what the code is doing. For example, `% Normalize data`.
- Surround variable names with backticks (``` `` ```) to identify variables to the Live Editor and prevent them from being renamed. If backticks are not added and the task has the same input and output variable name, then the input variable could get auto-incremented, which could result in errors.
- Minimize the amount of generated code by only including commands that set parameters to nondefault values. For example, this code checks whether the selected normalization method is the default method.
- Optionally, separate out the plotting code and add it to the end of the generated code.

```
function [code,outputs] = generateCode(task)
    if isequal(task.InputDataDropDown.Value,"select variable")
        % Not have enough information to generate code,
        % return empty values
        code = "";
        outputs = {};
        return
    end

    outputs = {'normalizedData'};

    code = "% Normalize data";
    code = code + newline + outputs{1} + " = normalize(";
    code = code + "`" + task.InputDataDropDown.Value + "`";
```

```
    if ~isequal(task.MethodDropDown.Value,"zscore") || ...
            ~isequal(task.ZscoreDropDown.Value,"std")
        code = code + ", """ + task.MethodDropDown.Value + """";
        if isequal(task.MethodDropDown.Value,"zscore")
            code = code + ", """ + task.ZscoreDropDown.Value + """";
        end
    end

    code = code + ");" + newline;
end
```

Implement reset Method

Define the `reset` method for your class to bring the task back to its default state. Define the method in the same method block as the `get.Summary`, `get.State`, `set.State`, and `generateCode` methods.

For example, implement the `reset` method for the `NormalizeVectorData` class. Call the `setComponentsToDefault` function to set all objects to their default values. Then, call the `updateComponents` function to update all the components.

```
function reset(task)
    setComponentsToDefault(task);
    updateComponents(task);
end
```

Implement postExecutionUpdate Method (Optional)

Optionally, you can define a `postExecutionUpdate` method for your class to perform specific updates for your task. This method executes after the generated code for your task runs. For example, you can implement the `postExecutionUpdate` method to add newly created variables to a drop-down list used to select input data after the generated code for your task runs.

Define the `postExecutionUpdate` method in the same method block as the `get.Summary`, `get.State`, `set.State`, `generateCode`, and `reset` methods. The `postExecutionUpdate` method takes two inputs, an instance of the task, and a struct containing the task outputs (returned by the `generateCode` method) and their current workspace values.

```
function postExecutionUpdate(task,data)
    ...
end
```

Emit StateChanged Event (Optional)

The Live Editor listens for changes in a task and calls the `generateCode` method to update the task's generated code when it detects a change. The Live Editor detects changes by monitoring the components in the task that fire these events:

- `ValueChanged`
- `ButtonPushed`
- `ImageClicked`
- `SelectionChanged`

To update the generated code for a task when changes occur outside of the events listed (for example, in a component that does not fire these events), you can call the `notify` method to fire the `StateChanged` event and trigger a call to the `generateCode` method for the task.

```
notify(task,"StateChanged");
```

Note The Live Editor does not monitor events for components that are created dynamically at run time.

Set AutoRun Property (Optional)

Optionally, you can set the AutoRun property for your class to specify whether your task runs automatically when a user modifies the task parameters.

By default, the AutoRun property is set to true and the task runs automatically after a change. To disable running your task automatically, set the AutoRun property to false.

```
task.AutoRun = false;
```

You only need to set the AutoRun property once, preferably when setting the initial state of the task. For example, set the AutoRun property for the NormalizeVectorData class in the setup method.

```
function setup(task)
    task.AutoRun = false;
    createComponents(task);
    setComponentsToDefault(task);
    updateComponents(task);
end
```

Configure Live Editor Task Metadata

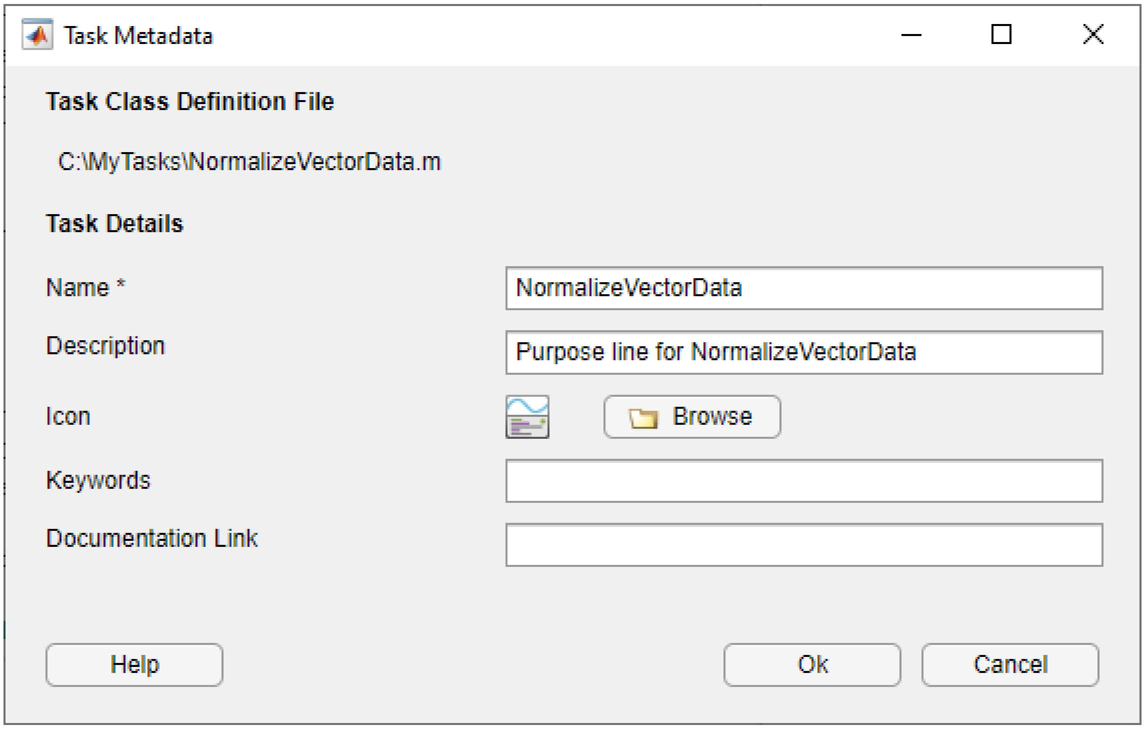
After creating the Live Editor task subclass, configure your task for use in the Live Editor by using the matlab.task.configureMetadata function. The function opens the Task Metadata dialog box. This dialog box allows you to specify metadata for the task. The Live Editor then uses this metadata to display the task in the Live Editor task gallery as well as in automatic code suggestions and completions.

Call the function by passing it the path to the class definition file for your task. If you do not specify a path, a file selection dialog box opens and prompts you to select a file.

For example, use the matlab.task.configureMetadata function to configure the Normalize Vector Data task.

```
matlab.task.configureMetadata("NormalizeVectorData")
```

The Task Metadata dialog box prepopulates all of the required task metadata details from your task class definition file.



Task Class Definition File

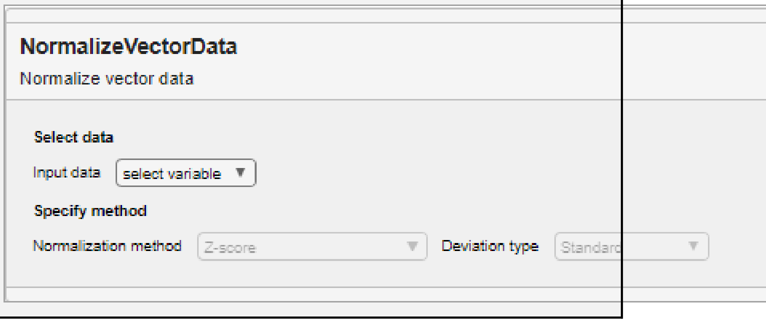
C:\MyTasks\NormalizeVectorData.m

Task Details

Name *	NormalizeVectorData
Description	Purpose line for NormalizeVectorData
Icon	Browse
Keywords	
Documentation Link	

You can edit the prepopulated metadata options using the Task Metadata dialog box. This table describes each individual task metadata option.

Option	Summary
Name	Specify the name of the task to display in the autocompletion list and in the Live Editor task gallery. This detail is required.
Description	Specify the description of the task to display in the autocompletion list and in the Live Editor task gallery. This detail is optional.
Icon	Specify the path for the task icon to display in the Live Editor task gallery. If you specify a task icon, MATLAB copies it to the resources folder. This detail is optional.
Keywords	Specify the keywords that can be used to show the task in the autocompletion list. This detail is optional.

Option	Summary
Documentation Link	Specify the documentation link as a URL to the documentation that opens when the task help icon is clicked. To specify a function that returns a documentation link dynamically, include the text `taskmethod:` followed by the name of the function that returns the doc link. For example, `taskmethod:fetchHelpLink`. The function must be defined as a public method in the task class definition file. If a documentation link is not specified, clicking the task help icon opens the generated help for the task in the Help browser. This detail is optional.  NormalizeVectorData Normalize vector data Select data Input data select variable ▼ Specify method Normalization method Z-score ▼ Deviation type Standar ▼

When you are done editing, select **OK**. MATLAB creates a folder named `resources` inside the folder containing your task class definition file. Inside the `resources` folder, MATLAB generates a file named `liveTasks.json`. This file contains the metadata you provided in the Task Metadata dialog box, in addition to other metadata MATLAB needs to make your task available in the Live Editor. Share this folder when you share your Live Editor task.

Note Do not modify the `liveTasks.json` file by hand. To change any Live Editor task metadata, use the `matlab.task.configureMetadata` function.

To make your task available in the Live Editor, add the folder containing your task class definition file to the MATLAB path. To add the folder, use the `addpath` function or the **Add Folder** button in the Set Path dialog box. To make your task available in the Live Editor in future MATLAB sessions, save the path using the `savepath` function or the **Save** button in the Set Path dialog box.

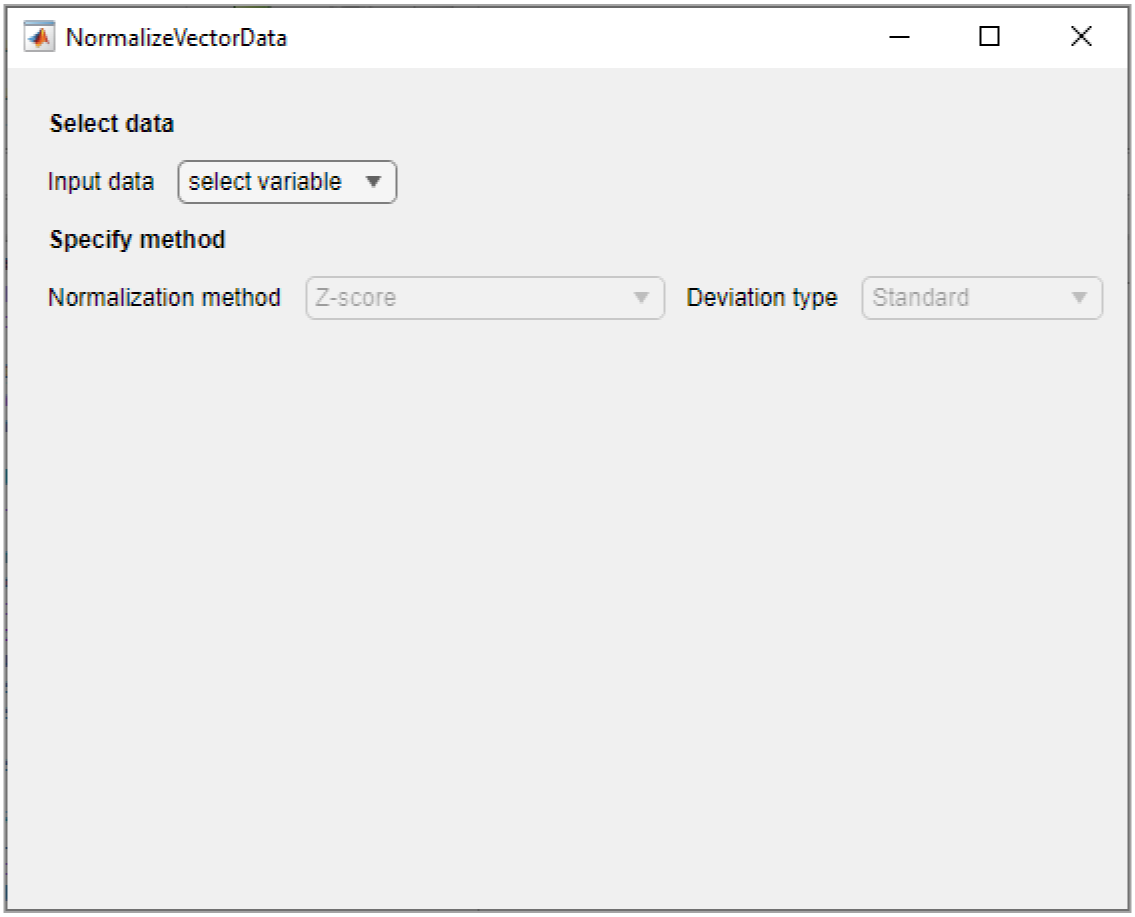
Note You do not need to add the `resources` folder that is inside the folder containing your task class definition file to the path. Folders named `resources` are not allowed on the MATLAB path.

Use Custom Live Editor Task

To test whether your task creates correctly, create an instance of your class. For example, create an instance of the `NormalizeVectorData` class.

```
c = NormalizeVectorData;
```

MATLAB creates the Normalize Vector Data task.



To get all of the UI objects in the task, use the `findall` function.

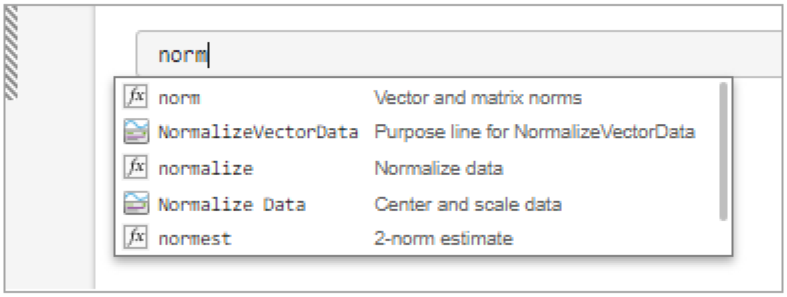
```
h = findall(c.Parent)
```

```
h = 
  14×1 graphics array:

  Figure        (NormalizeVectorData)
  GridLayout
  GridLayout
  Label         (Select data)
  GridLayout
  Label         (Specify method)
  GridLayout
  Label         (Input data)
  DropDown      (select variable)
```

```
Label          (Normalization method)
DropDown       (Z-score)
GridLayout
Label          (Deviation type)
DropDown       (Standard)
```

To add your task to a live script, in the live script, type one of the keywords defined in `liveTasks.json` and select your task from the list of suggested names.

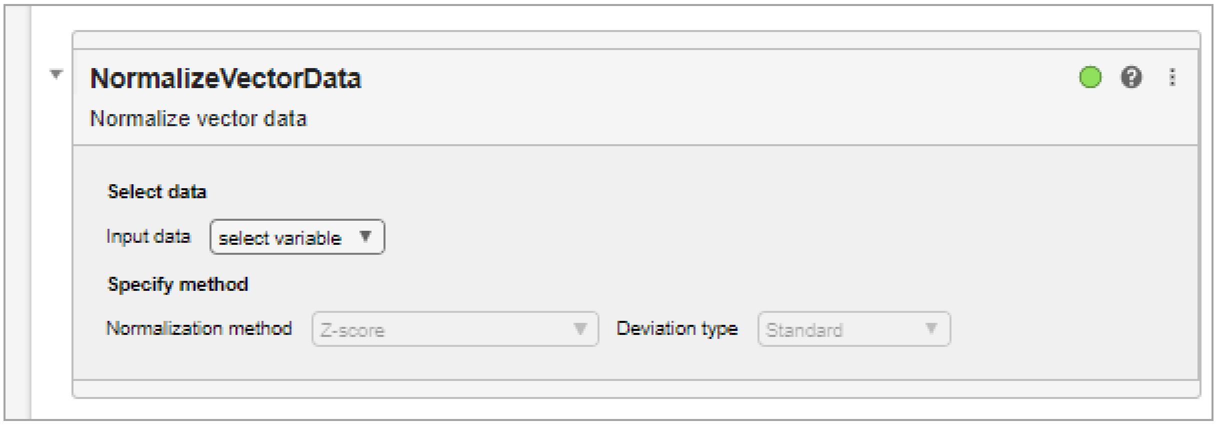
For example, to add the Normalize Vector Data task to a live script, first, create a live script. Then, on a code line, type `norm`. MATLAB shows a list of suggested matches.



norm

norm	Vector and matrix norms
NormalizeVectorData	Purpose line for NormalizeVectorData
normalize	Normalize data
Normalize Data	Center and scale data
normest	2-norm estimate

Select **Normalize Vector Data** from the list. MATLAB adds the Normalize Vector Data task to the live script.



NormalizeVectorData

Normalize vector data

Select data

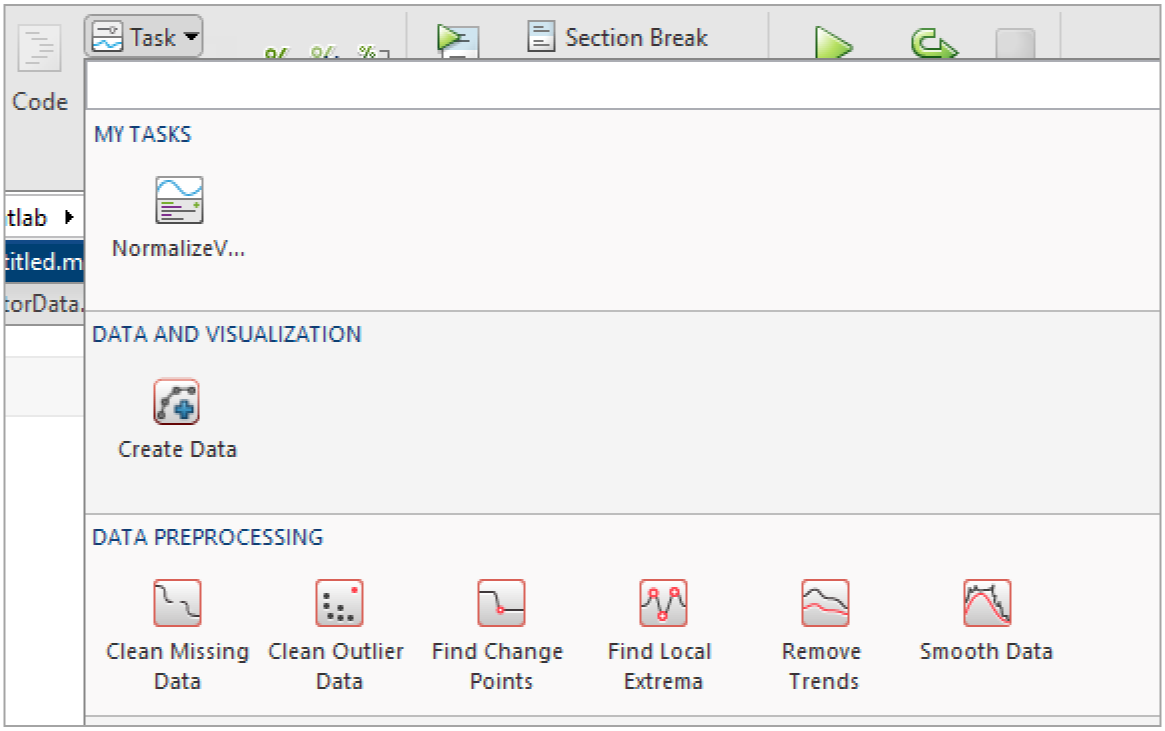
Input data: select variable

Specify method

Normalization method: Z-score

Deviation type: Standard

To add your task from the toolstrip, go to the **Live Editor** tab and in the **Code** section, click **Task** ▾. Select `Normalize Vector Data` to add the Normalize Vector Data task to your live script. You must save the MATLAB path and restart MATLAB to include your task in the Live Editor task gallery.



See Also

`matlab.task.LiveTask` | `setup` | `generateCode` | `reset`

Related Examples

- "Create Simple Live Editor Task" on page 16-14
- "Share Live Editor Tasks" on page 16-21

Create Simple Live Editor Task

This example shows how to create a simple custom Live Editor task and add it to a live script.

Define Live Editor Task Subclass

Define a class called NormalizeVectorData that creates a custom Live Editor task for normalizing vector data.

To define the class, create a file called NormalizeVectorData.m that contains the following class definition with these features:

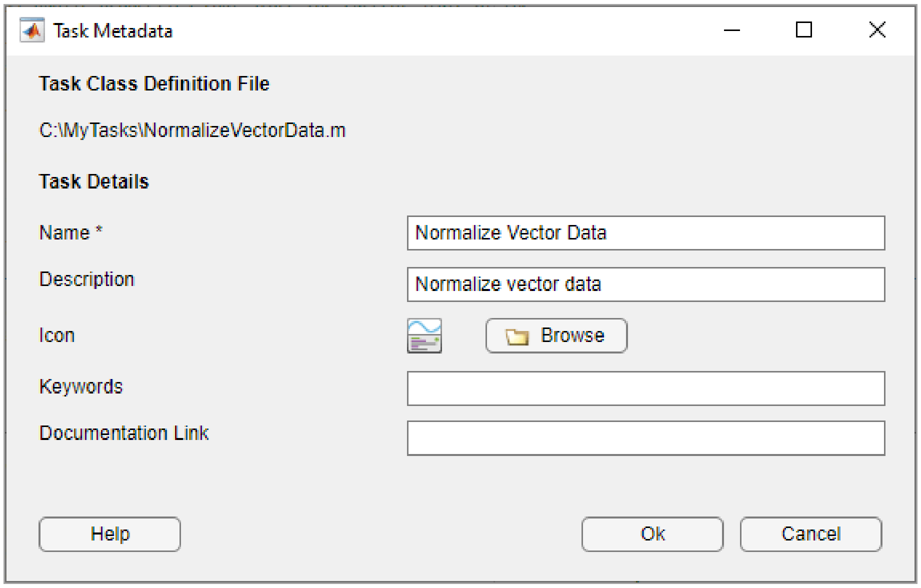
- State and Summary public properties that store the current state of the task and a dynamic summary of what the task does.
- Private properties that store the drop-down lists, spinners, checkboxes, and grid layout managers for selecting input data and specifying parameters.
- A setup method that initializes the task.
- A generateCode method that updates the generated code for the task.
- get.Summary, get.State, and set.State methods for getting and setting the summary and state of the task.
- An updateComponents method that updates the task when a user selects input data or changes parameters.
- A reset method that resets the state of the task.

```
classdef NormalizeVectorData < matlab.task.LiveTask
    properties(Access = private,Transient)
        InputDataDropDown               matlab.ui.control.DropDown
        MethodDropDown                  matlab.ui.control.DropDown
        ZscoreGrid                      matlab.ui.container.GridLayout
        ZscoreDropDown                  matlab.ui.control.DropDown
        RangeGrid                       matlab.ui.container.GridLayout
        LeftRangeSpinner                matlab.ui.control.Spinner
        RightRangeSpinner               matlab.ui.control.Spinner
        InputDataCheckBox               matlab.ui.control.CheckBox
        NormalizedDataCheckBox          matlab.ui.control.CheckBox
    end

    properties(Dependent)
        State
        Summary
    end

    methods(Access = private)
        function createComponents(task)
            g = uigridlayout(task.LayoutManager,[1,1]);
            g.RowHeight = {'fit' 'fit' 'fit' 'fit' 'fit' 'fit'};
            g.ColumnWidth = {'fit'};

            % Row 1: Select data section label
            uilabel(g,"Text","Select data","FontWeight","bold");

            % Row 2: Select data section components
            inputgrid = uigridlayout(g,"RowHeight",{'fit'},"ColumnWidth", ...
                {'fit','fit'},"Padding",0);
            uilabel(inputgrid,"Text","Input data");
            task.InputDataDropDown = uidropdown(inputgrid, ...
                "ValueChangedFcn",@task.updateComponents, ...
                "DropDownOpeningFcn",@task.populateWSDropdownItems);
            task.populateWSDropdownItems(task.InputDataDropDown);

            % Row 3: Specify method section label
            uilabel(g,"Text","Specify method","FontWeight","bold");

            % Row 4: Method section components
            methodgrid = uigridlayout(g,"RowHeight",{'fit'}, ...
                "ColumnWidth",{'fit','fit','fit'},"Padding",0);
            uilabel(methodgrid,"Text","Normalization method");
            task.MethodDropDown = uidropdown(methodgrid,"ValueChangedFcn", ...
                @task.updateComponents);
            task.MethodDropDown.Items = ["Z-score" "2-Norm" ...
                "Scale by standard deviation" "Scale to new range" ...
                "Center to mean 0"];
            task.MethodDropDown.ItemsData = {'zscore' 'norm' 'scale' ...
                'range' 'center'};

            % Subgrid 1 in method section
            task.ZscoreGrid = uigridlayout(methodgrid,"RowHeight",{'fit'}, ...
                "ColumnWidth",{'fit','fit'},"Padding",0);
            uilabel(task.ZscoreGrid,"Text","Deviation type");
            task.ZscoreDropDown = uidropdown(task.ZscoreGrid,"ValueChangedFcn", ...
                @task.updateComponents,"Items",{'Standard' 'Median absolute'}, ...
                "ItemsData",{'std' 'robust'},"Tooltip", ...
                "Center data to 0 and scale to deviation 1");

            % Subgrid 2 in method section
            task.RangeGrid = uigridlayout(methodgrid,"RowHeight",{'fit'}, ...
                "ColumnWidth",{'fit' 50 50},"Padding",0);
            task.RangeGrid.Layout.Row = 1;
            task.RangeGrid.Layout.Column = 3;
            uilabel(task.RangeGrid,"Text","Range edges");
            task.LeftRangeSpinner = uispinner(task.RangeGrid,"ValueChangedFcn", ...
                @task.updateComponents,"Tag","LeftRangeSpinner","Tooltip", ...
                "Left edge of new range");
            task.RightRangeSpinner = uispinner(task.RangeGrid,"ValueChangedFcn", ...
                @task.updateComponents,"Tag","RightRangeSpinner", ...
                "Tooltip","Right edge of new range");

            % Row 5: Display results section label
            uilabel(g,"Text","Display results","FontWeight","bold");

            % Row 6: Display results section components
            displaygrid = uigridlayout(g,"RowHeight",{'fit'},"ColumnWidth", ...
                {'fit','fit'},"Padding",0);
            task.InputDataCheckBox = uicheckbox(displaygrid,"Text", ...
                "Input data","ValueChangedFcn",@task.updateComponents);
            task.NormalizedDataCheckBox = uicheckbox(displaygrid,"Text", ...
                "Normalized data","ValueChangedFcn",@task.updateComponents);
        end

        function setComponentsToDefault(task)
```

```
            task.MethodDropDown.Value = "zscore";
            task.ZscoreDropDown.Value = "std";
            task.LeftRangeSpinner.Value = 0;
            task.RightRangeSpinner.Value = 1;
            task.InputDataCheckBox.Value = true;
            task.NormalizedDataCheckBox.Value = true;
        end

        function updateComponents(task,source,~)
            if nargin > 1
                if isequal(source.Tag,"LeftRangeSpinner")
                    if task.RightRangeSpinner.Value <= task.LeftRangeSpinner.Value
                        task.RightRangeSpinner.Value = task.LeftRangeSpinner.Value + 1;
                    end
                elseif isequal(source.Tag,"RightRangeSpinner")
                    if task.RightRangeSpinner.Value <= task.LeftRangeSpinner.Value
                        task.LeftRangeSpinner.Value = task.RightRangeSpinner.Value - 1;
                    end
                end
            end
            hasData = ~isequal(task.InputDataDropDown.Value,"select variable");
            task.MethodDropDown.Enable = hasData;
            task.ZscoreDropDown.Enable = hasData;
            task.LeftRangeSpinner.Enable = hasData;
            task.RightRangeSpinner.Enable = hasData;
            task.InputDataCheckBox.Enable = hasData;
            task.NormalizedDataCheckBox.Enable = hasData;
            % Show only relevant subgrids
            task.ZscoreGrid.Visible = isequal(task.MethodDropDown.Value,"zscore");
            task.RangeGrid.Visible = isequal(task.MethodDropDown.Value,"range");
            % Trigger the live editor to update the generated script
            notify(task,"StateChanged");
        end

        function populateWSDropdownItems(~,src,~)
            workspaceVariables = evalin("base","who");
            src.Items = ["select variable"; workspaceVariables];
            src.ItemsData = ["select variable"; workspaceVariables];
        end
    end

    methods(Access = protected)
        function setup(task)
            createComponents(task);
            setComponentsToDefault(task);
            updateComponents(task);
        end
    end

    methods
        function [code,outputs] = generateCode(task)
            if isequal(task.InputDataDropDown.Value,"select variable")
                % Not have enough information to generate code,
                % return empty values
                code = "";
                outputs = {};
                return
            end
            outputs = {'normalizedData'};

            code = "% Normalize data";
            code = code + newline + outputs{1} + " = normalize(";
            code = code + "`" + task.InputDataDropDown.Value + "`";

            if ~isequal(task.MethodDropDown.Value,"zscore") || ...
                    ~isequal(task.ZscoreDropDown.Value,"std")
                code = code + ", """ + task.MethodDropDown.Value + """";
                if isequal(task.MethodDropDown.Value,"zscore")
                    code = code + ", """ + task.ZscoreDropDown.Value + """";
                elseif isequal(task.MethodDropDown.Value,"range") && ...
                        (task.LeftRangeSpinner.Value ~= 0 || ...
                        task.RightRangeSpinner.Value ~= 1)
                    code = code + ",[" + num2str(task.LeftRangeSpinner.Value) ...
                        + ", " + num2str(task.RightRangeSpinner.Value) + "]";
                end
            end

            code = code + ");" + newline + newline + "% Visualize results" + ...
                newline + "figure" + newline;

            if task.InputDataCheckBox.Value
                code = code + "plot(`" + task.InputDataDropDown.Value + ...
                    "`, ""Color"",[109 185 226]/255," + ...
                    " ""DisplayName"",""Input data"")";
            end

            if task.NormalizedDataCheckBox.Value
                if task.InputDataCheckBox.Value
                    code = code + newline + "hold on" + newline;
                end
                code = code + "plot(normalizedData,""Color"",[0 114 189]/255, ..." + ...
                    newline + " ""LineWidth"",1.5,""DisplayName"",""Normalized data"")";
                if task.InputDataCheckBox.Value
                    code = code + newline + "hold off";
                end
            end

            code = code + newline + "legend";
        end

        function summary = get.Summary(task)
            if isequal(task.InputDataDropDown.Value,"select variable")
                summary = "Normalize vector data";
            else
                switch task.MethodDropDown.Value
                    case "zscore"
                        methodString = " using z-score";
                    case "norm"
                        methodString = " using 2-norm";
                    case "scale"
                        methodString = " using scaling by standard deviation";
                    case "range"
                        methodString = " by scaling to new range";
                    case "center"
                        methodString = " by centering the data to 0";
                end
                summary = "Normalized vector `" + task.InputDataDropDown.Value + ...
                    "`" + methodString;
            end
        end

        function state = get.State(task)
            state = struct;
            state.InputDataDropDownValue = task.InputDataDropDown.Value;
            state.MethodDropDownValue = task.MethodDropDown.Value;
            state.ZscoreDropDownValue = task.ZscoreDropDown.Value;
            state.LeftRangeSpinnerValue = task.LeftRangeSpinner.Value;
            state.RightRangeSpinnerValue = task.RightRangeSpinner.Value;
            state.InputDataCheckboxValue = task.InputDataCheckBox.Value;
            state.NormalizedDataCheckboxValue = task.NormalizedDataCheckBox.Value;
        end

        function set.State(task,state)
            value = state.InputDataDropDownValue;
            if ~ismember(value, task.InputDataDropDown.ItemsData)
                % In case the selected Input Data variable was cleared after
                % saving the mlx file and before reopening the mlx file
                task.InputDataDropDown.Items = [task.InputDataDropDown.Items {value}];
                task.InputDataDropDown.ItemsData = [task.InputDataDropDown.ItemsData {value}];
            end
            task.InputDataDropDown.Value = value;
            task.MethodDropDown.Value = state.MethodDropDownValue;
            task.ZscoreDropDown.Value = state.ZscoreDropDownValue;
            task.LeftRangeSpinner.Value = state.LeftRangeSpinnerValue;
            task.RightRangeSpinner.Value = state.RightRangeSpinnerValue;
            task.InputDataCheckBox.Value = state.InputDataCheckboxValue;
            task.NormalizedDataCheckBox.Value = state.NormalizedDataCheckboxValue;
            updateComponents(task);
        end

        function reset(task)
            setComponentsToDefault(task);
            updateComponents(task);
        end
    end
end
```

Configure Live Editor Task Metadata

To configure the task metadata, call the `matlab.task.configureMetadata` function and select the `NormalizeVectorData.m` file. The Task Metadata dialog box opens with all of the required task metadata details prepopulated.



Task Metadata

Task Class Definition File

C:\MyTasks\NormalizeVectorData.m

Task Details

Name *: Normalize Vector Data

Description: Normalize vector data

Icon

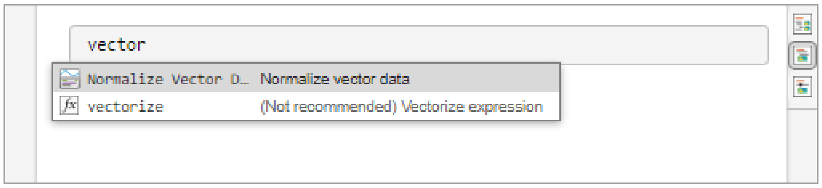
Keywords:

Documentation Link:

Select **OK** to use the prepopulated metadata details. MATLAB creates a folder named `resources` inside the folder containing your task class definition file. Inside the `resources` folder, MATLAB generates a file named `liveTasks.json`. Add the folder containing the task class definition file to the MATLAB path by calling the `addpath` function or using the **Add Folder** button in the Set Path dialog box. To make your task available in the Live Editor in future MATLAB sessions, save the path by calling the `savepath` function or using the **Save** button in the Set Path dialog box.

Add Live Editor Task to Live Script

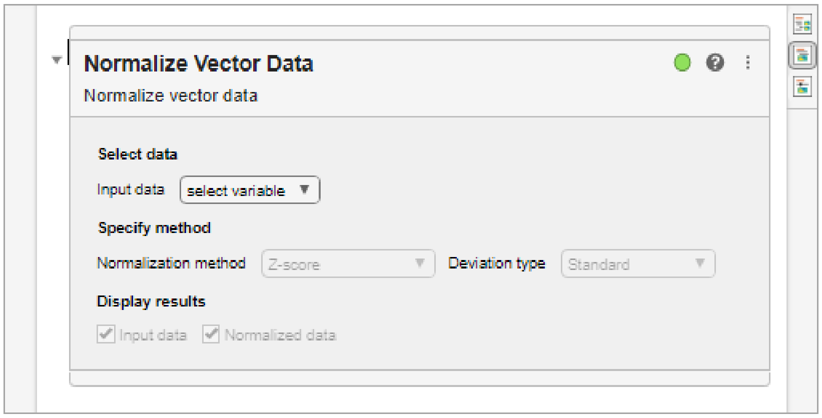
On a code line, type `vector`. MATLAB shows a list of suggested matches.



```
vector
```

Normalize Vector D…	Normalize vector data
vectorize	(Not recommended) Vectorize expression

Select **Normalize Vector Data** from the list. MATLAB adds the Normalize Vector Data task to the live script.



Normalize Vector Data

Normalize vector data

Select data

Input data: select variable ▼

Specify method

Normalization method: Z-score ▼ Deviation type: Standard ▼

Display results

☑ Input data ☑ Normalized data

See Also

`matlab.task.LiveTask` | `setup` | `generateCode` | `reset`

Related Examples

- "Live Editor Task Development Overview" on page 16-2
- "Share Live Editor Tasks" on page 16-21

Share Live Editor Tasks

After creating your own Live Editor task, you can share the task for others to use in the Live Editor.

To share a Live Editor task with other users, create and share a folder with these contents:

- The Live Editor task class definition file
- The generated `resources` folder containing the `liveTasks.json` file

Instruct the users you are sharing the Live Editor task with to add the shared folder to the MATLAB path. To add the folder, they can use the `addpath` function or the **Add Folder** button in the Set Path dialog box. To make the task available in the Live Editor in future MATLAB sessions, they must also save the path using the `savepath` function or the **Save** button in the Set Path dialog box.

Note The `resources` folder does not need to be added to the path. Folders named `resources` are not allowed on the MATLAB path.

Once the shared folder is added to the path, users must restart MATLAB. Then, they can see your Live Editor task in the Live Editor task gallery as well as in automatic code suggestions and completions.

See Also

Classes
`matlab.task.LiveTask`

Functions
`savepath` | `addpath`

Related Examples

- “Live Editor Task Development Overview” on page 16-2
- “Create Simple Live Editor Task” on page 16-14

Create UIs with GUIDE

GUIDE Preferences and Options

GUIDE Preferences

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see "GUIDE Migration Strategies" on page 3-7 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, "Develop Apps Using App Designer" instead.

Set Preferences

You can set preferences for GUIDE. From the MATLAB **Home** tab, in the **Environment** section, click **Preferences**. These preferences apply to GUIDE and to all UIs you create.

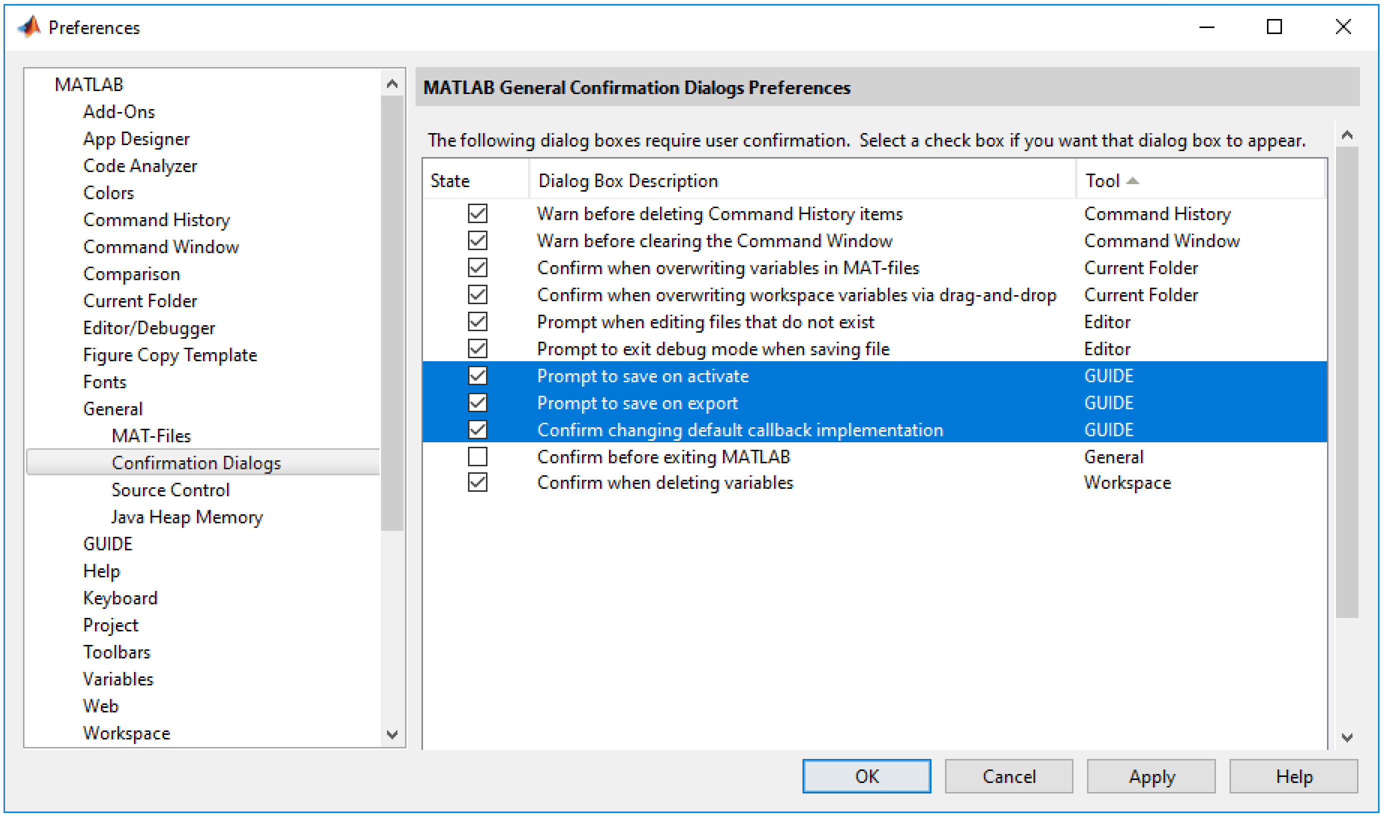
The preferences are in different locations within the Preferences dialog box:

Confirmation Preferences

GUIDE provides two confirmation preferences. You can choose whether you want to display a confirmation dialog box when you:

- Activate a UI from GUIDE.
- Export a UI from GUIDE.
- Change a callback signature generated by GUIDE.

In the Preferences dialog box, click **MATLAB > General > Confirmation Dialogs** to access the GUIDE confirmation preferences. Look for the word `GUIDE` in the **Tool** column.



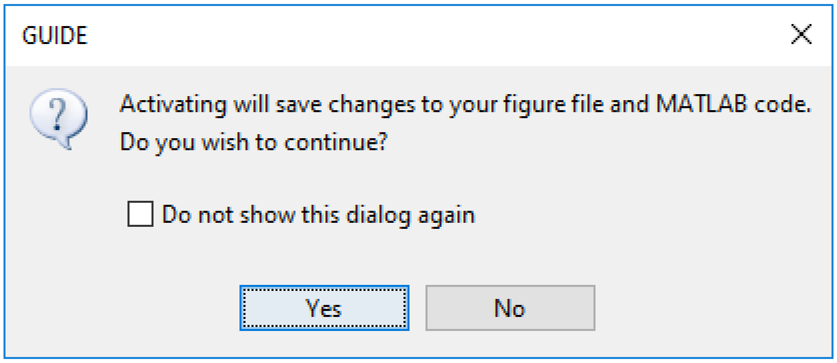
MATLAB General Confirmation Dialogs Preferences

The following dialog boxes require user confirmation. Select a check box if you want that dialog box to appear.

State	Dialog Box Description	Tool
☑	Warn before deleting Command History items	Command History
☑	Warn before clearing the Command Window	Command Window
☑	Confirm when overwriting variables in MAT-files	Current Folder
☑	Confirm when overwriting workspace variables via drag-and-drop	Current Folder
☑	Prompt when editing files that do not exist	Editor
☑	Prompt to exit debug mode when saving file	Editor
☑	Prompt to save on activate	GUIDE
☑	Prompt to save on export	GUIDE
☑	Confirm changing default callback implementation	GUIDE
□	Confirm before exiting MATLAB	General
☑	Confirm when deleting variables	Workspace

Prompt to Save on Activate

When you activate a UI from the Layout Editor by clicking the **Run** button ▶, a dialog box informs you of the impending save and lets you choose whether or not you want to continue.

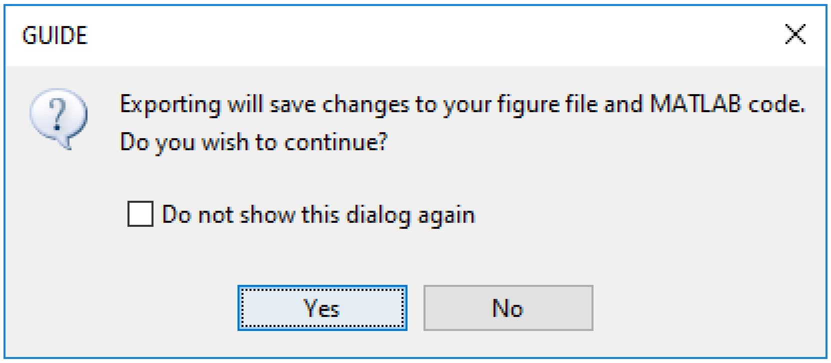


Activating will save changes to your figure file and MATLAB code. Do you wish to continue?

□ Do not show this dialog again

Prompt to Save on Export

From the Layout Editor, when you select **File > Export to MATLAB-file**, a dialog box informs you of the impending save and lets you choose whether or not you want to continue.



GUIDE

Exporting will save changes to your figure file and MATLAB code. Do you wish to continue?

□ Do not show this dialog again

Backward Compatibility Preference

MATLAB Version 5 or Later Compatibility

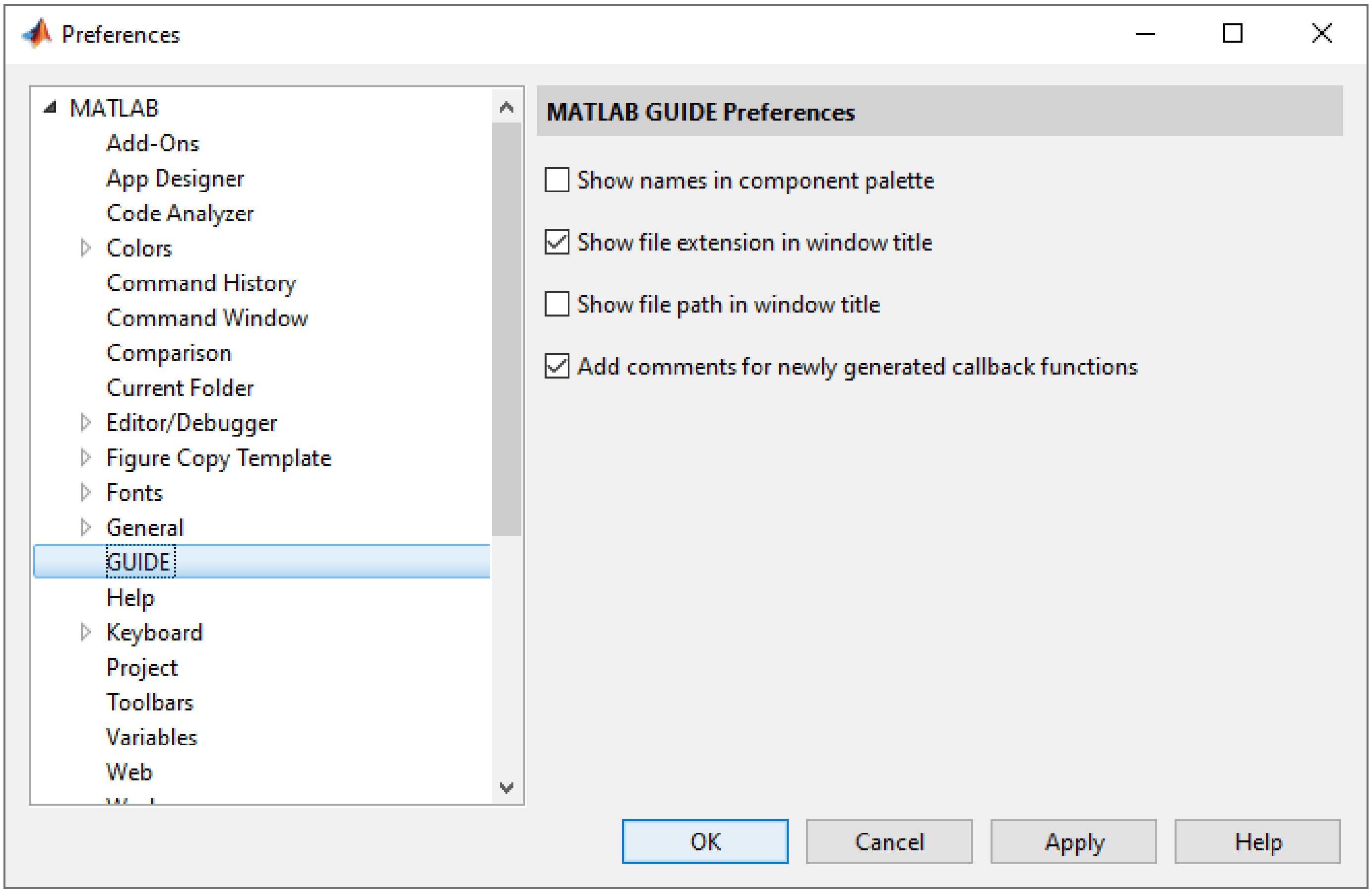
UI FIG-files created or modified with MATLAB 7.0 or a later version are not automatically compatible with Version 6.5 and earlier versions. GUIDE automatically generates FIG-files, which are binary files that contain the UI layout information.

To make a FIG-file backward compatible, from the Layout Editor, select **File** > **Preferences** > **General** > **MAT-Files**, and then select **MATLAB Version 5 or later (save -v6)**.

Note The **-v6** option discussed in this section is obsolete and will be removed in a future version of MATLAB.

All Other Preferences

GUIDE provides other preferences, for the Layout Editor interface and for inserting code comments. In the Preferences dialog box, click **GUIDE** to access these preferences.



MATLAB GUIDE Preferences

□ Show names in component palette

☑ Show file extension in window title

□ Show file path in window title

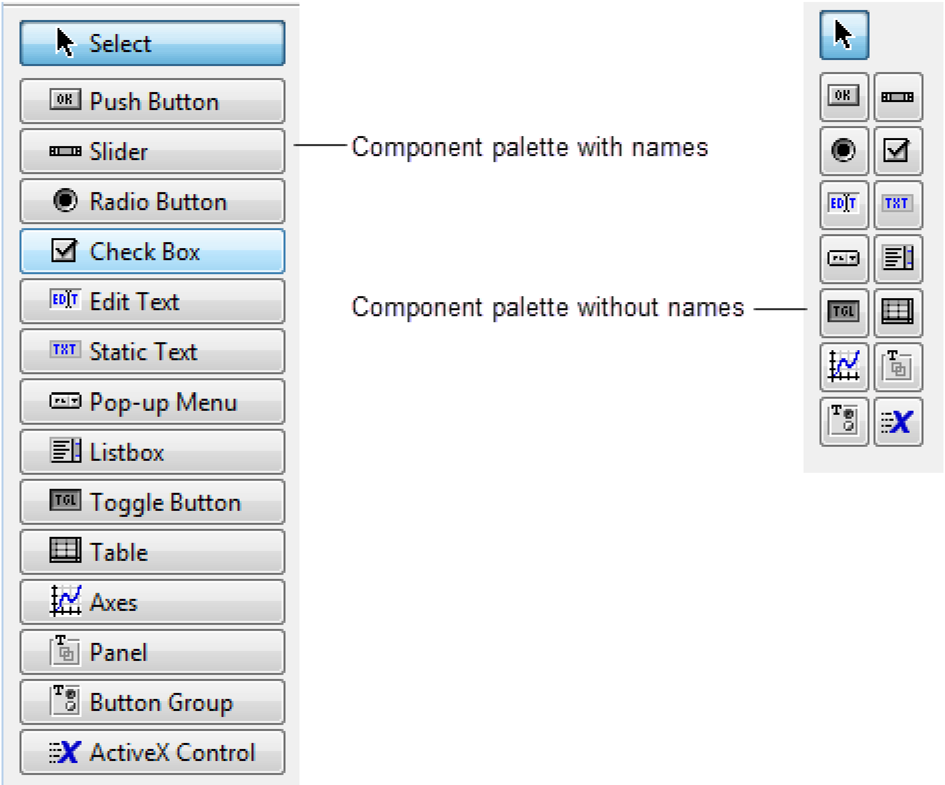
☑ Add comments for newly generated callback functions

The following topics describe the preferences in this dialog:

- “Show Names in Component Palette” on page 17-5
- “Show File Extension in Window Title” on page 17-6
- “Show File Path in Window Title” on page 17-6
- “Add Comments for Newly Generated Callback Functions” on page 17-6

Show Names in Component Palette

Displays both icons and names in the component palette, as shown below. When unchecked, the icons alone are displayed in two columns, with tooltips.



Show File Extension in Window Title

Displays the FIG-file file name with its file extension, `.fig`, in the Layout Editor window title. If unchecked, only the file name is displayed.

Show File Path in Window Title

Displays the full file path in the Layout Editor window title. If unchecked, the file path is not displayed.

Add Comments for Newly Generated Callback Functions

Callbacks are blocks of code that execute in response to actions by the user, such as clicking buttons or manipulating sliders. By default, GUIDE sets up templates that declare callbacks as functions and adds comments at the beginning of each one. Most of the comments are similar to the following.

```
% --- Executes during object deletion, before destroying properties.
function figure1_DeleteFcn(hObject, eventdata, handles)
% hObject    handle to figure1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

Some callbacks are added automatically because their associated components are part of the original GUIDE template that you chose. Other commonly used callbacks are added automatically when you add components. You can also add callbacks explicitly by selecting them from **View > View Callbacks** menu or on the component's context menu.

If you deselect this preference, GUIDE includes comments only for callbacks that are automatically included to support the original GUIDE template. GUIDE does not include comments for callbacks subsequently added to the code.

See “Write Callbacks in GUIDE” on page 19-2 for more information about callbacks and about the arguments described in the preceding comments.

See Also

Related Examples

- “GUIDE Options” on page 17-8
- “GUIDE Migration Strategies” on page 3-7

GUIDE Options

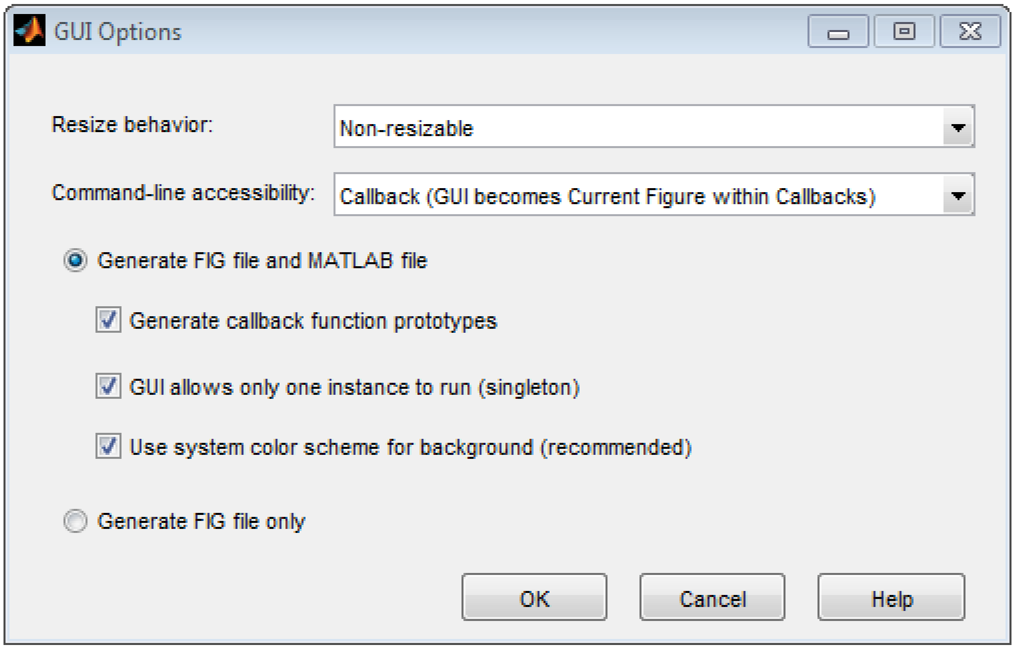
In this section...

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see "GUIDE Migration Strategies" on page 3-7 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, "Develop Apps Using App Designer" instead.

The GUI Options Dialog Box

Access the dialog box from the GUIDE Layout Editor by selecting **Tools** > **GUI Options**. The options you select take effect the next time you save your UI.



Resize behavior: Non-resizable

Command-line accessibility: Callback (GUI becomes Current Figure within Callbacks)

● Generate FIG file and MATLAB file

☑ Generate callback function prototypes

☑ GUI allows only one instance to run (singleton)

☑ Use system color scheme for background (recommended)

○ Generate FIG file only

Resize Behavior

You can control whether users can resize the window and how MATLAB handles resizing. GUIDE provides three options:

- **Non-resizable** — Users cannot change the window size (default).
- **Proportional** — The software automatically scales the components in the UI in proportion to the new figure window size.
- **Other (Use SizeChangedFcn)** — Program the UI to behave in a certain way when users resize the figure window.

The first two options set figure and component properties appropriately and require no other action. **Other (Use SizeChangedFcn)** requires you to write a callback routine that recalculates sizes and positions of the components based on the new figure size.

Command-Line Accessibility

You can restrict access to a figure window from the command line or from a code file with the GUIDE **Command-line accessibility** options.

Unless you explicitly specify a figure handle, many commands, such as `plot`, alter the current figure (the figure specified by the root `CurrentFigure` property and returned by the `gcf` command). The current figure is usually the figure that is most recently created, drawn into, or mouse-clicked. You can programmatically designate a figure `h` (where `h` is its handle) as the current figure in four ways:

1. `set(groot,'CurrentFigure',h)` — Makes figure `h` current, but does not change its visibility or stacking with respect to other figures
2. `figure(h)` — Makes figure `h` current, visible, and displayed on top of other figures
3. `axes(h)` — Makes existing axes `h` the current axes and displays the figure containing it on top of other figures
4. `plot(h,...)`, or any plotting function that takes an axes as its first argument, also makes existing axes `h` the current axes and displays the figure containing it on top of other figures

The `gcf` function returns the handle of the current figure.

```
h = gcf
```

For a UI created in GUIDE, set the **Command-line accessibility** option to prevent users from inadvertently changing the appearance or content of a UI by executing commands at the command line or from a script or function, such as `plot`. The following table briefly describes the four options for **Command-line accessibility**.

Option	Description
Callback (GUI becomes Current Figure within Callbacks)	The UI can be accessed only from within a callback. The UI cannot be accessed from the command line or from a script. This is the default.
Off (GUI never becomes Current Figure)	The UI cannot be accessed from a callback, the command line, or a script, without the handle.
On (GUI may become Current Figure from Command Line)	The UI can be accessed from a callback, from the command line, and from a script.
Other (Use settings from Property Inspector)	You control accessibility by setting the `HandleVisibility` and `IntegerHandle` properties from the Property Inspector.

Generate FIG-File and MATLAB File

Select **Generate FIG-file and MATLAB file** in the **GUI Options** dialog box if you want GUIDE to create both the FIG-file and the UI code file (this is the default). Once you have selected this option, you can select any of the following items in the frame to configure UI code:

- “Generate Callback Function Prototypes” on page 17-10
- “GUI Allows Only One Instance to Run (Singleton)” on page 17-10
- “Use System Color Scheme for Background” on page 17-11

See “Files Generated by GUIDE” on page 2-2 for information about these files.

Generate Callback Function Prototypes

If you select **Generate callback function prototypes** in the **GUI Options** dialog, GUIDE adds templates for the most commonly used callbacks to the code file for most components. You must then insert code into these templates.

GUIDE also adds a callback whenever you edit a callback routine from the Layout Editor's right-click context menu and when you add menus to the UI using the Menu Editor on page 18-41.

See “Write Callbacks in GUIDE” on page 19-2 for general information about callbacks.

Note This option is available only if you first select the **Generate FIG-file and MATLAB file** option.

GUI Allows Only One Instance to Run (Singleton)

This option allows you to select between two behaviors for the figure window:

- Allow MATLAB software to display only one instance of the UI at a time.
- Allow MATLAB software to display multiple instances of the UI.

If you allow only one instance, the software reuses the existing figure whenever the command to run your program is executed. If a UI window already exists, the software brings it to the foreground rather than creating a new figure.

If you clear this option, the software creates a new figure whenever you issue the command to run the program.

Even if you allow only one instance of a UI to exist, initialization can take place each time you invoke it from the command line. For example, the code in an `OpeningFcn` will run each time a GUIDE program runs unless you take steps to prevent it from doing so. Adding a flag to the `handles` structure is one way to control such behavior. You can do this in the `OpeningFcn`, which can run initialization code if this flag doesn't yet exist and skip that code if it does.

Note This option is available only if you first select the **Generate FIG-file and MATLAB file** option.

Use System Color Scheme for Background

The default color used for UI components is system dependent. This option enables you to make the figure background color the same as the default component background color.

To ensure that the figure background matches the color of the components, select **Use system color scheme for background** in the **GUI Options** dialog.

Note This option is available only if you first select the **Generate FIG-file and MATLAB file** option.

Generate FIG-File Only

The **Generate FIG-file only** option enables you to open figures and UIs to perform limited editing. These can be any figures and need not be UIs. UIs need not have been generated using GUIDE. This mode provides limited editing capability and may be useful for UIs generated in MATLAB Versions 5.3 and earlier. See the `guide` function for more information.

GUIDE selects **Generate FIG-file only** as the default if you do one of the following:

- Start GUIDE from the command line by providing one or more figure objects as arguments.

  ```
  guide(f)
  ```

  In this case, GUIDE selects **Generate FIG-file only**, even when a code file with a corresponding name exists in the same folder.
- Start GUIDE from the command line and provide the name of a FIG-file for which no code file with the same name exists in the same folder.

  ```
  guide('myfig.fig')
  ```
- Use the GUIDE **Open Existing GUI** tab to open a FIG-file for which no code file with the same name exists in the same folder.

When you save the figure or UI with **Generate FIG-file only** selected, GUIDE saves only the FIG-file. You must update any corresponding code files yourself, as appropriate.

If you want GUIDE to manage the UI code file for you, change the selection to **Generate FIG-file and MATLAB file** before saving the UI. If there is no corresponding code file in the same location, GUIDE creates one. If a code file with the same name as the original figure or UI exists in the same folder, GUIDE overwrites it. To prevent overwriting an existing file, save the UI using **Save As** from the **File** menu. Select another file name for the two files. GUIDE updates variable names in the new code file as appropriate.

Callbacks for UIs without Code

Even when there is no code file associated with a UI FIG-file, you can still provide callbacks for UI components to make them perform actions when used. In the Property Inspector, you can type callbacks in the form of character vectors, built-in functions, or MATLAB code file names; when your program runs, it will execute them if possible. If the callback is a file name, it can include arguments to that function. For example, setting the `Callback` property of a push button to `sqrt(2)` causes the result of the expression to display in the Command Window:

```
ans =
    1.4142
```

Any file that a callback executes must be in the current folder or on the MATLAB path. For more information on how callbacks work, see “Write Callbacks in GUIDE” on page 19-2

See Also

Related Examples

- “GUIDE Preferences” on page 17-2

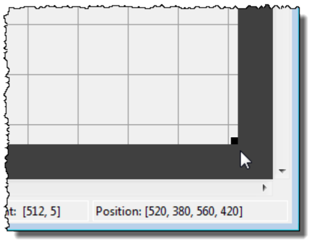
Lay Out a UI Using GUIDE

Set the UI Window Size in GUIDE

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 3-7 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

Set the size of the UI window by resizing the grid area in the Layout Editor. Click the lower-right corner of the layout area and drag it until the UI is the desired size. If necessary, make the window larger.



t: [512, 5]

Position: [520, 380, 560, 420]

As you drag the corner handle, the readout in the lower right corner shows the current position of the UI in pixels.

Setting the `Units` property to `characters` (nonresizable UIs) or `normalized` (resizable UIs) gives the UI a more consistent appearance across platforms.

Prevent Existing Objects from Resizing with the Window

Existing objects within the UI resize with the window if their `Units` are set to `'normalized'`. To prevent them from resizing with the window, perform these steps:

1. Set each object’s `Units` property to an absolute value, such as inches or pixels before enlarging the UI.

   To change the `Units` property for all the objects in your UI simultaneously, drag a selection box around all the objects, and then click the Property Inspector button and set the `Units`.
2. When you finish enlarging the UI, set each object’s `Units` property back to `normalized`.

Set the Window Position or Size to an Exact Value

1. In the Layout Editor, open the Property Inspector for the figure by clicking the button (with no components selected).
2. In the Property Inspector, scroll to the `Units` property and note whether the current setting is `characters` or `normalized`.
3. Click the down arrow at the far right in the `Units` row, and select `inches`.
4. In the Property Inspector, display the `Position` property elements by clicking the **+** sign to the left of `Position`.

5 Change the `x` and `y` coordinates to the point where you want the lower-left corner of the window to appear, and its width and height.

6 Reset the `Units` property to its previous setting, as noted in step 2.

Maximize the Layout Area

You can make maximum use of space within the Layout Editor by hiding the GUIDE toolbar and status bar, and showing only tool icons, as follows:

1 From the **View** menu, deselect **Show Toolbar**.

2 From the **View** menu, deselect **Show Status Bar**.

3 Select **File** > **Preferences**, and then clear **Show names in component palette**

See Also

Related Examples

- "Ways to Build Apps" on page 1-2
- "GUIDE Options" on page 17-8

Add Components to the GUIDE Layout Area

In this section...

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 3-7 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

Place Components

The component palette at the left side of the Layout Editor contains the components that you can add to your UI.

To place components in the GUIDE layout area and give each component a unique identifier, follow these steps:

1 Display component names on the palette.

   a On the MATLAB **Home** tab, in the **Environment** section, click **Preferences**.

   b In the Preferences dialog box, click **GUIDE**.

   c Select **Show Names in Component Palette**, and then click **OK** .

2 Place components in the layout area according to your design.

   - Drag a component from the palette and drop it in the layout area.
   - Click a component in the palette and move the cursor over the layout area. The cursor changes to a cross. Click again to add the component in its default size, or click and drag to size the component as you add it.

   Once you have defined a UI component in the layout area, selecting it automatically shows it in the Property Inspector. If the Property Inspector is not open or is not visible, double-clicking a component raises the inspector and focuses it on that component.

   The components listed in the following table have additional considerations; read more about them in the sections described there.

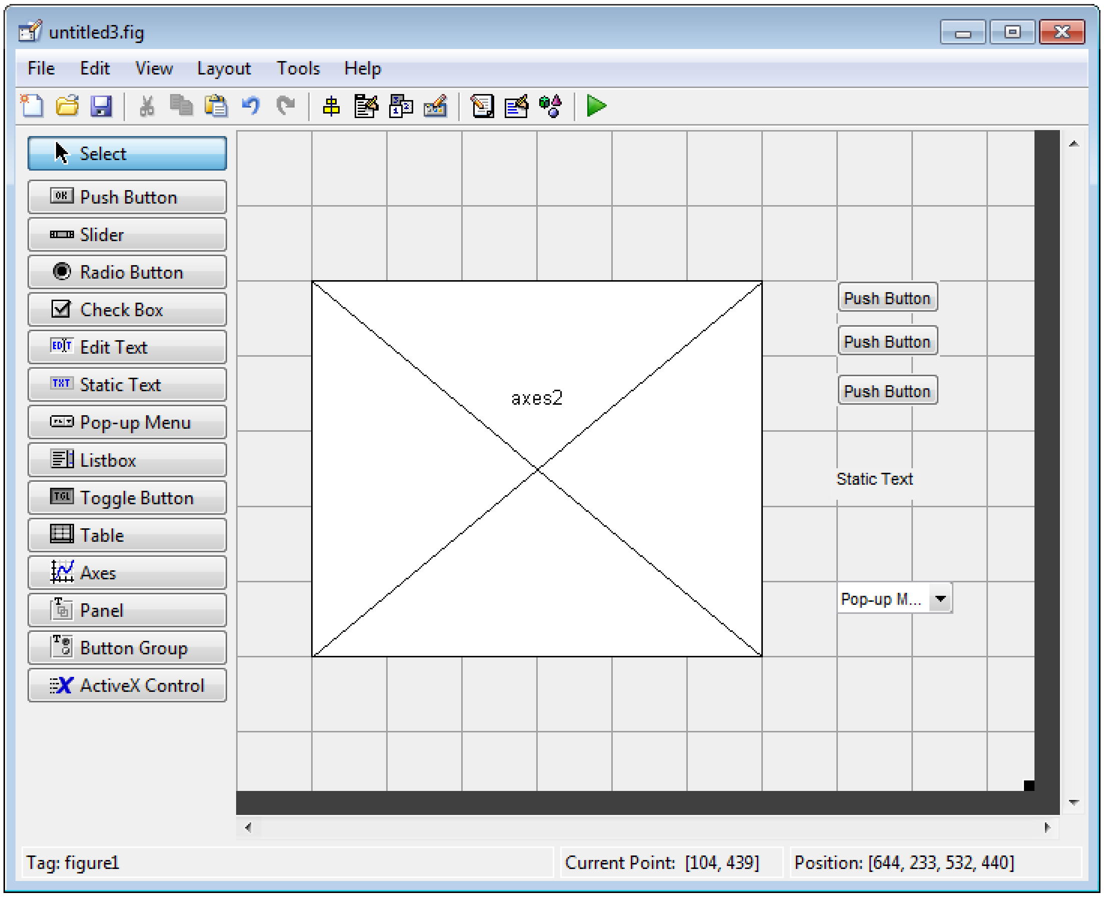
If You Are Adding...	Then...
Panels or button groups	See “Add a Component to a Panel or Button Group” on page 18-6.
Menus	See “Create Menus for GUIDE Apps” on page 18-41

3 Assign a unique identifier to each component. Do this by setting the value of the component `Tag` properties. See “Assign an Identifier to Each Component” on page 18-8 for more information.

4 Specify the look and feel of each component by setting the appropriate properties. The following topics contain specific information.

- “User Interface Controls” on page 18-9
- “Panels and Button Groups” on page 18-23
- “Axes” on page 18-27
- “Table” on page 18-30

This is an example of a UI in the Layout Editor. Components in the Layout Editor are not active.



Tag: figure1

Current Point: [104, 439]

Position: [644, 233, 532, 440]

Use Coordinates to Place Components

The status bar at the bottom of the GUIDE Layout Editor displays:

- **Current Point** — The current location of the mouse relative to the lower left corner of the grid area in the Layout Editor.
- **Position** — The `Position` property of the selected component is a vector: [distance from left, distance from bottom, width, height], where distances are relative to the parent figure, panel, or button group.

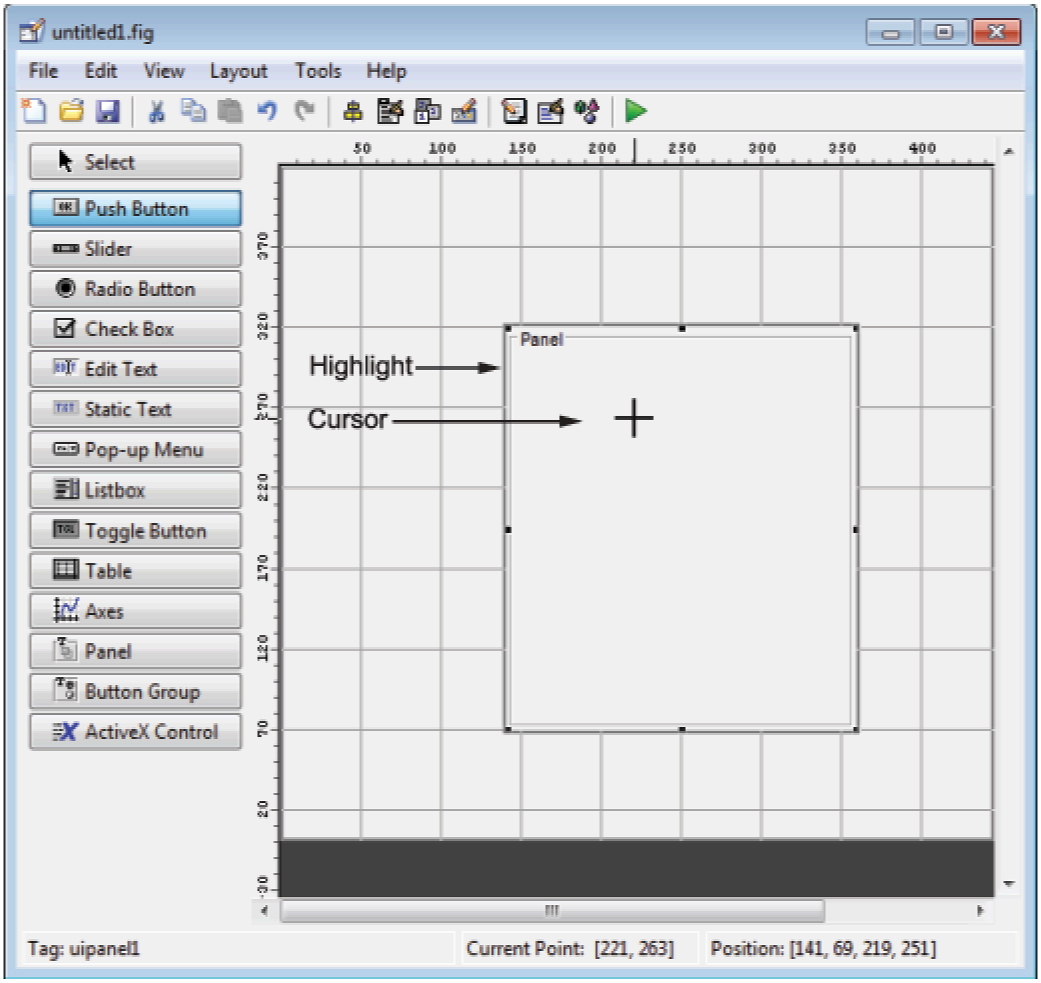
Here is how to interpret the coordinates in the status bar and rulers:

- The **Position** values updates as you move and resize components. The first two elements in the vector change as you move the component. The last two elements of the vector change as the height and width of the component change.
- When no components are selected, the **Position** value displays the location and size of the figure.

Add a Component to a Panel or Button Group

To add a component to a panel or button group, select the component in the component palette then move the cursor over the desired panel or button group. The position of the cursor determines the component's parent.

GUIDE highlights the potential parent as shown in the following figure. The highlight indicates that if you drop the component or click the cursor, the component will be a child of the highlighted panel, button group, or figure.



Assign a unique identifier to each component in your panel or button group by setting the value of its `Tag` property. See “Assign an Identifier to Each Component” on page 18-8 for more information.

Include Existing Components in Panels and Button Groups

When you add a new component or drag an existing component to a panel or button group, it will become a member, or child, of the panel or button group automatically, whether fully or partially enclosed by it. However, if the component is not entirely contained in the panel or button group, it appears to be clipped in the Layout Editor and in the running app.

You can add a new panel or button group to a UI in order to group any of its existing controls. In order to include such controls in a new panel or button group, do the following. The instructions refer to panels, but you do the same for components inside button groups.

1. Select the New Panel or New Button Group tool and drag out a rectangle to have the size and position you want.

   The panel will not obscure any controls within its boundary unless they are axes, tables, or other panels or button groups. Only overlap panels you want to nest, and then make sure the overlap is complete.

2 You can use **Send Backward** or **Send to Back** on the **Layout** menu to layer the new panel behind components you do not want it to obscure, if your layout has this problem. As you add components to it or drag components into it, the panel will automatically layer itself behind them.

   Now is a good time to set the panel's `Tag` and `String` properties to whatever you want them to be, using the Property Inspector.

3 Open the Object Browser from the **View** menu and find the panel you just added. Use this tool to verify that it contains all the controls you intend it to group together. If any are missing, perform the following steps.

4 Drag controls that you want to include but don't fit within the panel inside it to positions you want them to have. Also, slightly move controls that are already in their correct positions to group them with the panel.

   The panel highlights when you move a control, indicating it now contains the control. The Object Browser updates to confirm the relationship. If you now move the panel, its child controls move with it.

> **Tip** You need to move controls with the mouse to register them with the surrounding panel or button group, even if only by a pixel or two. Selecting them and using arrow keys to move them does not accomplish this. Use the Object Browser to verify that controls are properly nested.

See “Panels and Button Groups” on page 18-23 for more information on how to incorporate panels and button groups into a UI.

Assign an Identifier to Each Component

Use the `Tag` property to assign a unique and meaningful identifier to your components.

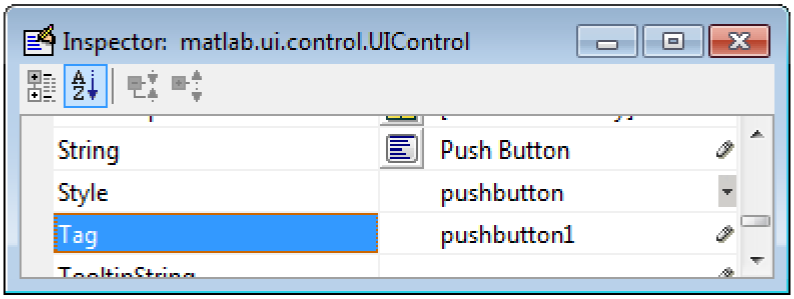
When you place a component in the layout area, GUIDE assigns a default value to the `Tag` property. Before saving the UI, replace this value with a name or abbreviation that reflects the role of the component in the UI.

The name you assign is used by code to identify the component and must be unique in the UI. To set the `Tag` property:

1 Select **View > Property Inspector** or click the **Property Inspector** button.

2 In the layout area, select the component for which you want to set `Tag`.

3 In the Property Inspector, select `Tag` and then replace the value with the name you want to use as the identifier. In the following figure, `Tag` is set to `pushbutton1`.



String	Push Button
Style	pushbutton
Tag	pushbutton1
TooltipString	

User Interface Controls

User interface controls include push buttons, toggle buttons, sliders, radio buttons, edit text controls, static text controls, pop-up menus, check boxes, and list boxes.

To define user interface controls, you must set certain properties. To do this:

1. Use the Property Inspector to modify the appropriate properties. Open the Property Inspector by selecting **View > Property Inspector** or by clicking the Property Inspector button.
2. In the layout area, select the component you are defining.

Subsequent topics describe commonly used properties of user interface controls and offer a simple example for each kind of control:

- “Commonly Used Properties” on page 18-9
- “Push Button” on page 18-10
- “Slider” on page 18-11
- “Radio Button” on page 18-12
- “Check Box” on page 18-13
- “Edit Text” on page 18-14
- “Static Text” on page 18-16
- “Pop-Up Menu” on page 18-17
- “List Box” on page 18-19
- “Toggle Button” on page 18-21

Commonly Used Properties

The most commonly used properties needed to describe a user interface control are shown in the following table. Instructions for a particular control may also list properties that are specific to that control.

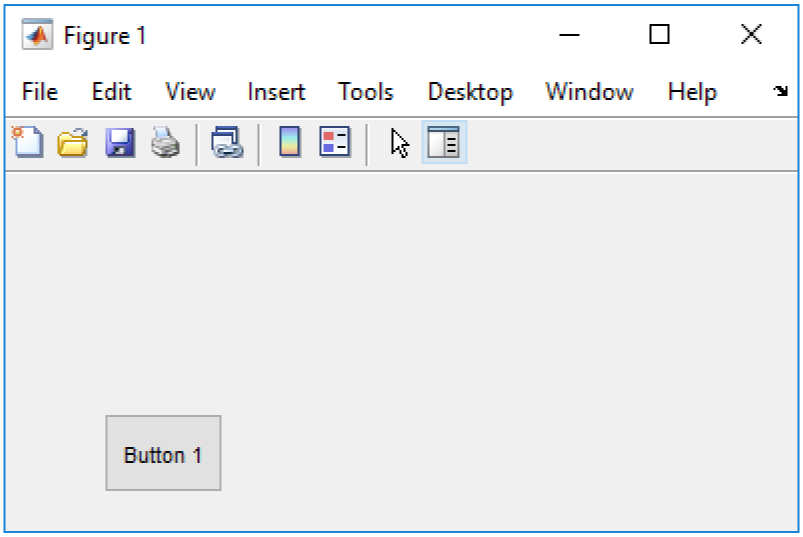
Property	Value	Description
`Enable`	`on`, `inactive`, `off`. Default is `on`.	Determines whether the control is available to the user
`Max`	Scalar. Default is 1.	Maximum value. Interpretation depends on the type of component.
`Min`	Scalar. Default is 0.	Minimum value. Interpretation depends on the type of component.
`Position`	4-element vector: [distance from left, distance from bottom, width, height].	Size of the component and its location relative to its parent.
`String`	Character vector (for example, `'button1'`). Can an also be a character array or a cell array of character vectors.	Component label. For list boxes and pop-up menus it is a list of the items.

Property	Value	Description
`Units`	`characters`, `centimeters`, `inches`, `normalized`, `pixels`, `points`. Default is `characters`.	Units of measurement used to interpret the `Position` property vector
`Value`	Scalar or vector	Value of the component. Interpretation depends on the type of component.

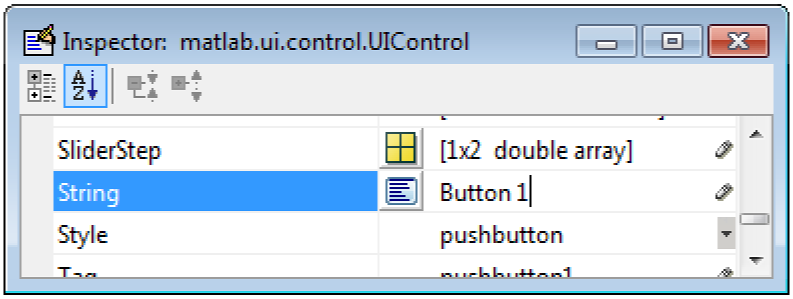
For a complete list of properties and for more information about the properties listed in the table, see Uicontrol.

Push Button

To create a push button with label **Button 1**, as shown in this figure:



- Specify the push button label by setting the `String` property to the desired label, in this case, `Button 1`.



SliderStep	[1x2 double array]
String	Button 1
Style	pushbutton
Tag	pushbutton1

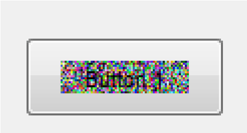
To display the & character in a label, use two & characters. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (`\`). For example, `\remove` yields **remove.**

The push button accommodates only a single line of text. If you specify more than one line, only the first line is shown. If you create a push button that is too narrow to accommodate the specified `String` property value, MATLAB truncates the value with an ellipsis.

- If you want to set the position or size of the component to an exact value, then modify its `Position` property.
- To add an image to a push button, assign the button's `CData` property as an m-by-n-by-3 array of RGB values that defines a truecolor image. You must do this programmatically in the opening function of the code file. For example, the array `img` defines a 16-by-64-by-3 truecolor image using random values between 0 and 1 (generated by `rand`).

```
img = rand(16,64,3);
set(handles.pushbutton1,'CData',img);
```

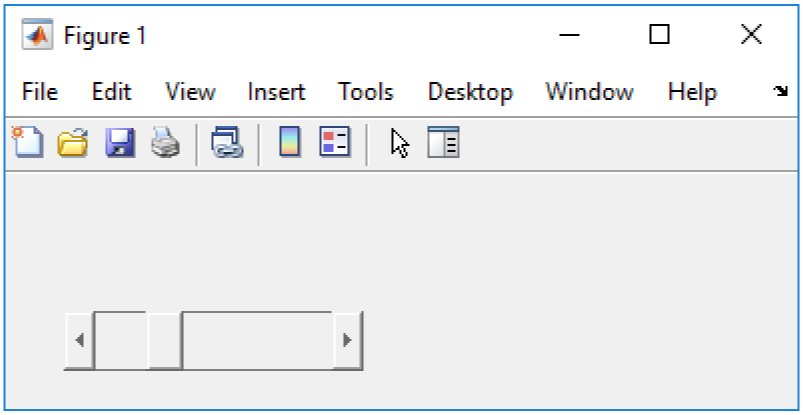
  where `pushbutton1` is the push button's `Tag` property.



  See `ind2rgb` for information on converting a matrix X and corresponding colormap, i.e., an `(X, MAP)` image, to RGB (truecolor) format.

Slider

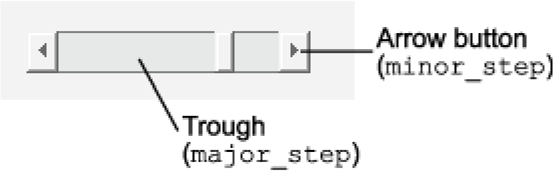
To create a slider as shown in this figure:



- Specify the range of the slider by setting its `Min` property to the minimum value of the slider and its `Max` property to the maximum value. The `Min` property must be less than `Max`.
- Specify the value indicated by the slider when it is created by setting the `Value` property to the appropriate number. This number must be less than or equal to `Max` and greater than or equal to `Min`. If you specify `Value` outside the specified range, the slider is not displayed.
- The slider `Value` changes by a small amount when a user clicks the arrow button, and changes by a larger amount when the user clicks the trough (also called the channel). Control how the slider responds to these actions by setting the `SliderStep` property. Specify `SliderStep` as a two-element vector, `[minor_step major_step]`, where `minor_step` is less than or equal to `major_step`. Because specifying very small values can cause unpredictable slider behavior, make

both minor_step and major_step greater than 1e-6. Set major_step to the proportion of the range that clicking the trough moves the slider thumb. Setting it to 1 or higher causes the thumb to move to Max or Min when the trough is clicked.

As major_step increases, the thumb grows longer. When major_step is 1, the thumb is half as long as the trough. When major_step is greater than 1, the thumb continues to grow, slowly approaching the full length of the trough. When a slider serves as a scroll bar, you can uses this behavior to indicate how much of the document is currently visible by changing the value of major_step.



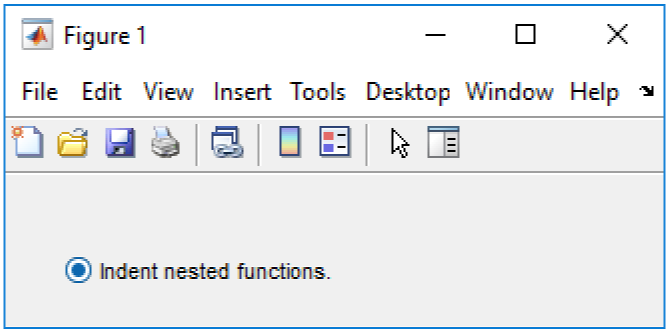
- If you want to set the location or size of the component to an exact value, then modify its Position property.

  The slider component provides no text description or data entry capability. Use a "Static Text" on page 18-16 component to label the slider. Use an "Edit Text" on page 18-14 component to enable a user to input a value to apply to the slider.

  On Mac platforms, the height of a horizontal slider is constrained. If the height you set in the position vector exceeds this constraint, the displayed height of the slider is the maximum allowed. The height element of the position vector is not changed.

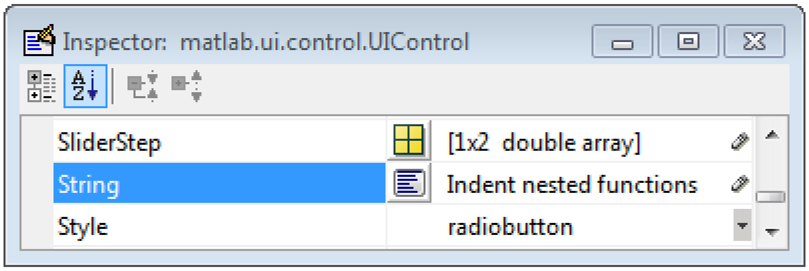
Radio Button

To create a radio button with label **Indent nested functions**, as shown in this figure:



● Indent nested functions.

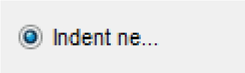
- Specify the radio button label by setting the String property to the desired label, in this case, Indent nested functions.



SliderStep	[1x2 double array]
String	Indent nested functions
Style	radiobutton

To display the & character in a label, use two & characters. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (`\`). For example, `\remove` yields **remove.**

The radio button accommodates only a single line of text. If you specify more than one line, only the first line is shown. If you create a radio button that is too narrow to accommodate the specified `String` property value, MATLAB software truncates the value with an ellipsis.



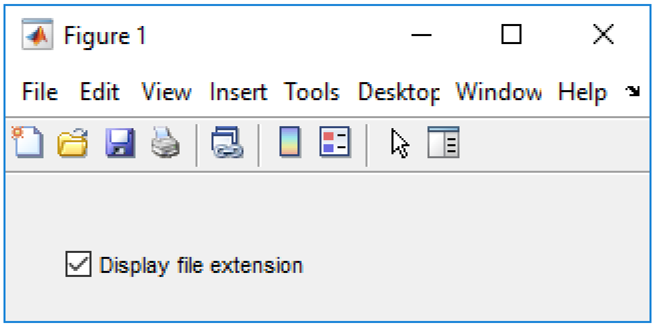
- Create the radio button with the button selected by setting its `Value` property to the value of its `Max` property (default is `1`). Set `Value` to `Min` (default is `0`) to leave the radio button unselected. Correspondingly, when the user selects the radio button, the software sets `Value` to `Max`, and to `Min` when the user deselects it.
- If you want to set the position or size of the component to an exact value, then modify its `Position` property.
- To add an image to a radio button, assign the button's `CData` property an m-by-n-by-3 array of RGB values that defines a truecolor image. You must do this programmatically in the opening function of the code file. For example, the array `img` defines a 16-by-24-by-3 truecolor image using random values between 0 and 1 (generated by `rand`).

  ```
  img = rand(16,24,3);
  set(handles.radiobutton1,'CData',img);
  ```

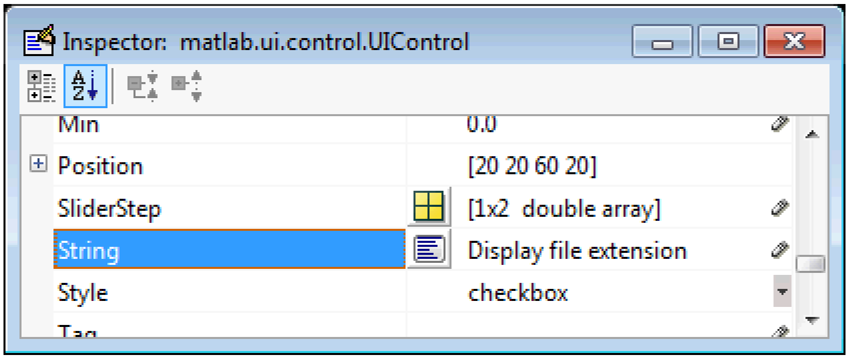
  To manage exclusive selection of radio buttons and toggle buttons, put them in a button group. See “Button Group” on page 18-25 for more information.

Check Box

To create a check box with label **Display file extension** that is initially checked, as shown in this figure:



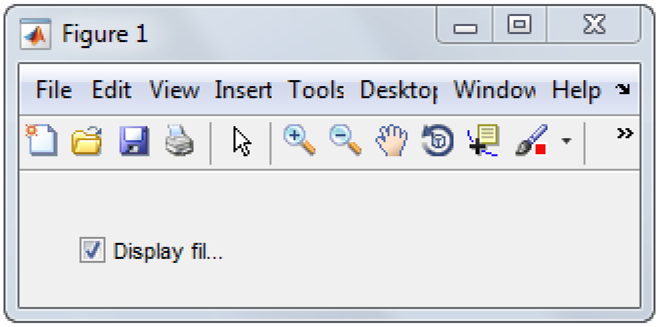
- Specify the check box label by setting the `String` property to the desired label, in this case, `Display file extension`.



Min	0.0
Position	[20 20 60 20]
SliderStep	[1x2 double array]
String	Display file extension
Style	checkbox
Tag	

To display the & character in a label, use two & characters. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (`\`). For example, `\remove` yields **remove.**

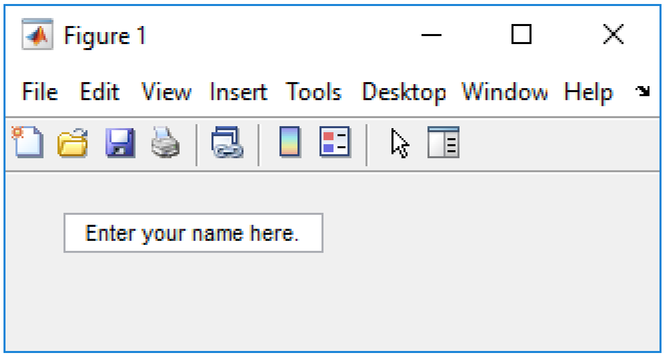
The check box accommodates only a single line of text. If you specify a component width that is too small to accommodate the specified `String` property value, MATLAB software truncates the value with an ellipsis.



- Create the check box with the box checked by setting the `Value` property to the value of the `Max` property (default is `1`). Set `Value` to `Min` (default is `0`) to leave the box unchecked. Correspondingly, when the user clicks the check box, the software sets `Value` to `Max` when the user checks the box and to `Min` when the user clears it.
- If you want to set the position or size of the component to an exact value, then modify its `Position` property.

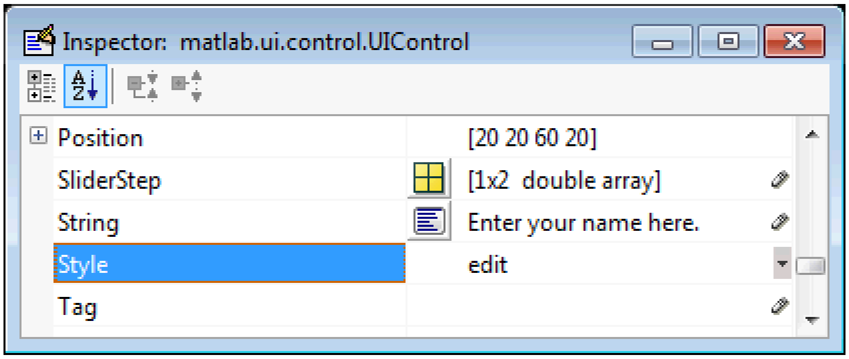
Edit Text

To create an edit text component that displays the initial text **Enter your name here**, as shown in this figure:



Enter your name here.

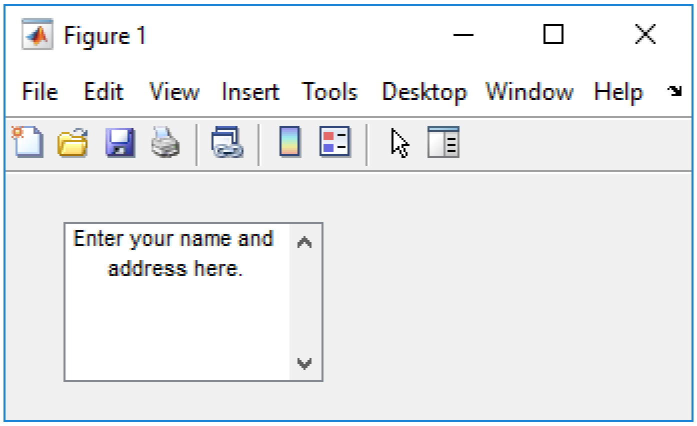
- Specify the text to be displayed when the edit text component is created by setting the `String` property to the desired value, in this case, `Enter your name here`.



⊞ Position	[20 20 60 20]
SliderStep	[1x2 double array]
String	Enter your name here.
Style	edit
Tag	

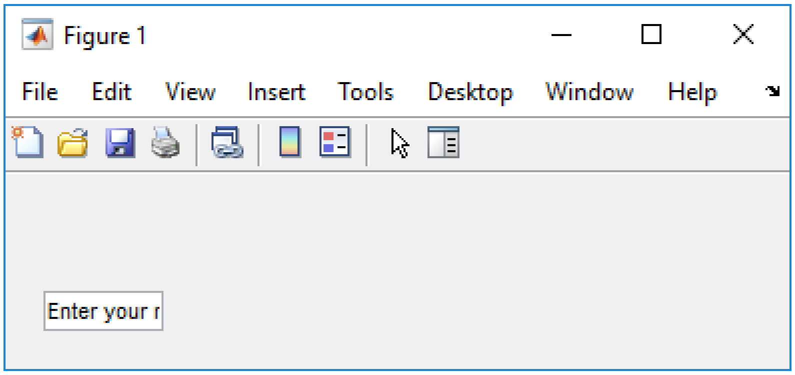
  To display the & character in a label, use two & characters. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (\). For example, `\remove` yields **remove.**

- To enable multiple-line input, specify the `Max` and `Min` properties so that their difference is greater than 1. For example, `Max = 2`, `Min = 0`. `Max` default is 1, `Min` default is 0. MATLAB software wraps the displayed text and adds a scroll bar if necessary. On all platforms, when the user enters a multiline text box via the **Tab** key, the editing cursor is placed at its previous location and no text highlights.



Enter your name and address here.

  If `Max-Min` is less than or equal to 1, the edit text component allows only a single line of input. If you specify a component width that is too small to accommodate the specified text, MATLAB displays only part of that text. The user can use the arrow keys to move the cursor through the text. On all platforms, when the user enters a single-line text box via the **Tab** key, the entire contents is highlighted and the editing cursor is at the end of the text.

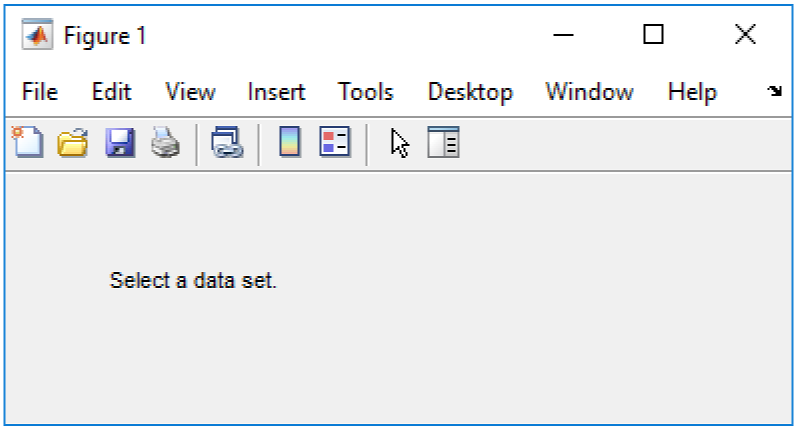


- If you want to set the position or size of the component to an exact value, then modify its `Position` property.
- You specify the text font to display in the edit box by typing the name of a font residing on your system into the `FontName` entry in the Property Inspector. On Microsoft® Windows platforms, the default is `MS Sans Serif`; on Macintosh and UNIX® platforms, the default is `Helvetica`.

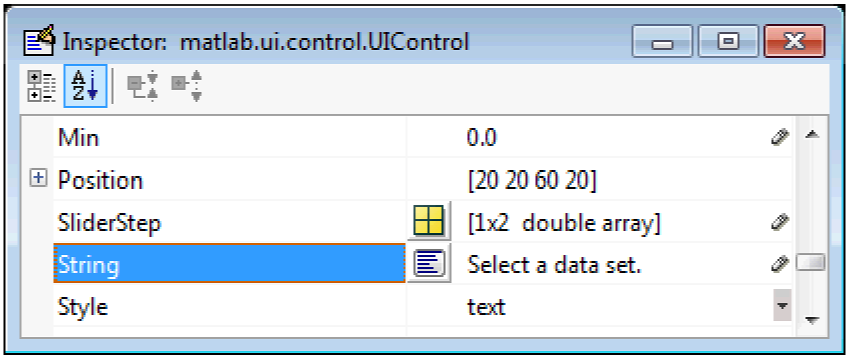
Tip To find out what fonts are available, type `uisetfont` at the MATLAB prompt; a dialog displays containing a list box from which you can select and preview available fonts. When you select a font, its name and other characteristics are returned in a structure, from which you can copy the `FontName` and paste it into the Property Inspector. Not all fonts listed may be available on other systems.

Static Text

To create a static text component with text **Select a data set**, as shown in this figure:



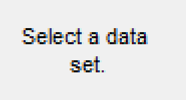
- Specify the text that appears in the component by setting the component `String` property to the desired text, in this case `Select a data set`.



Min	0.0
Position	[20 20 60 20]
SliderStep	[1x2 double array]
String	Select a data set.
Style	text

To display the & character in a list item, use two & characters. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (\). For example, `\remove` yields **remove.**

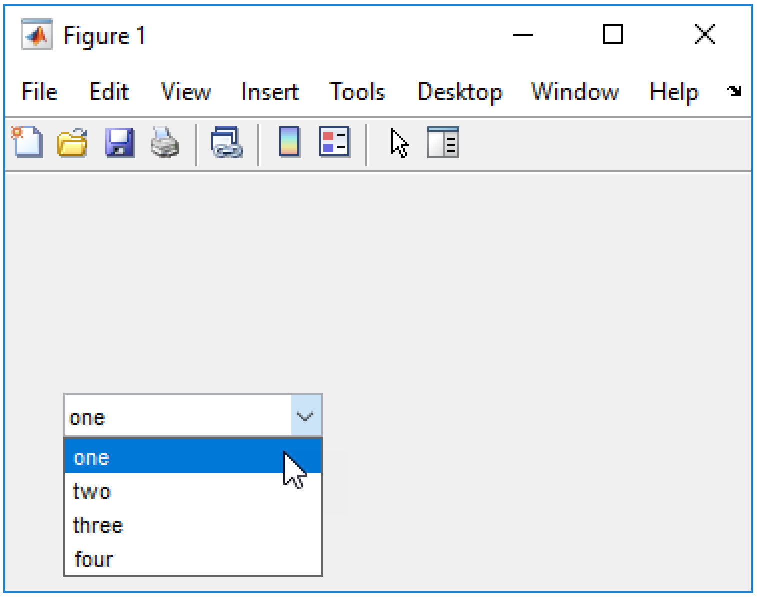
If your component is not wide enough to accommodate the specified value, MATLAB wraps the displayed text.



- If you want to set the position or size of the component to an exact value, then modify its `Position` property.
- You can specify a text font, including its `FontName`, `FontWeight`, `FontAngle`, `FontSize`, and `FontUnits` properties. For details, see the previous topic, “Edit Text” on page 18-14.

Pop-Up Menu

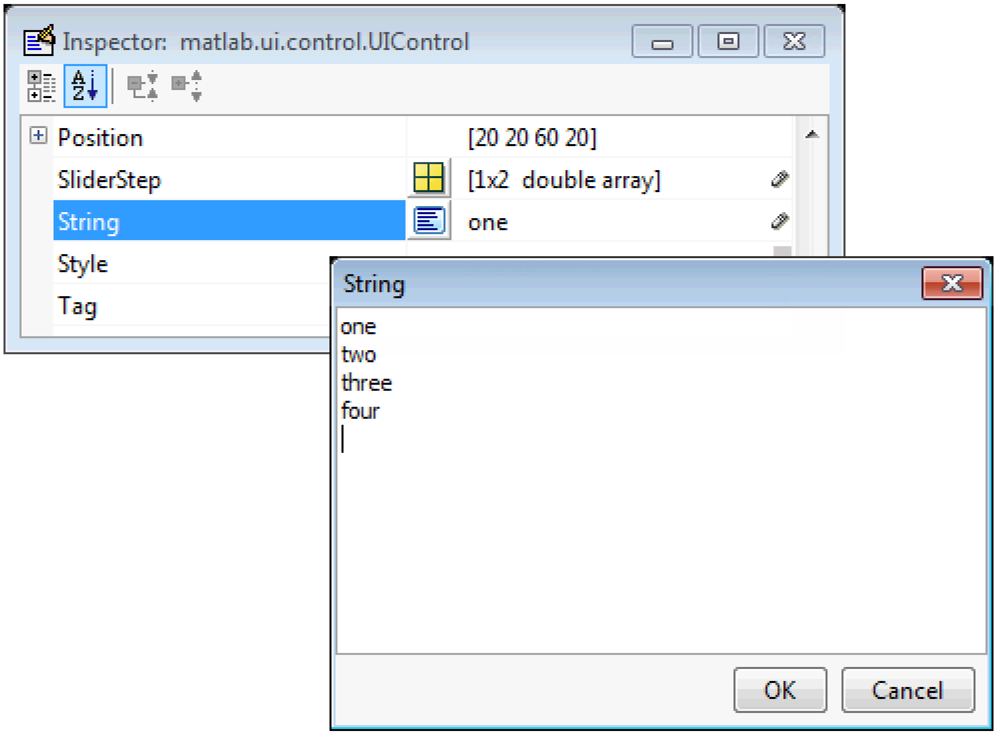
To create a pop-up menu (also known as a drop-down menu or combo box) with items **one**, **two**, **three**, and **four**, as shown in this figure:



- Specify the pop-up menu items to be displayed by setting the `String` property to the desired items. Click the

  ≣

  button to the right of the property name to open the Property Inspector editor.

  Inspector: matlab.ui.control.UIControl

Position	[20 20 60 20]
SliderStep	[1x2 double array]
String	one
Style	
Tag	

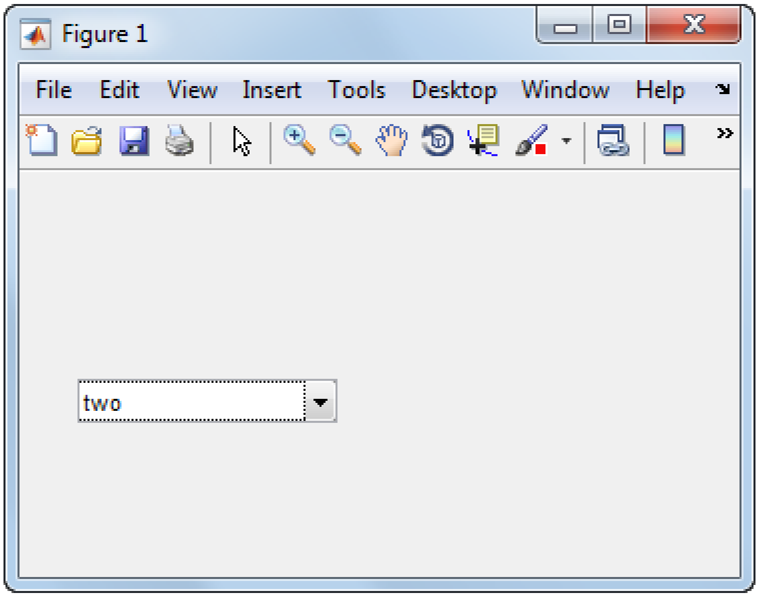
  String

  ```
  one
  two
  three
  four
  ```

  To display the & character in a menu item, use two & characters. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (\). For example, `\remove` yields **remove.**

  If the width of the component is too small to accommodate one or more of the menu items, MATLAB truncates those items with an ellipsis.

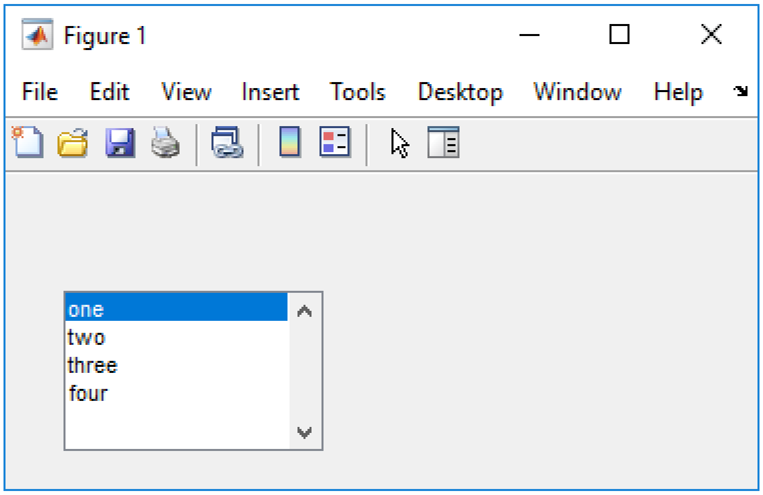
- To select an item when the component is created, set `Value` to a scalar that indicates the index of the selected list item, where 1 corresponds to the first item in the list. If you set `Value` to `2`, the menu looks like this when it is created:



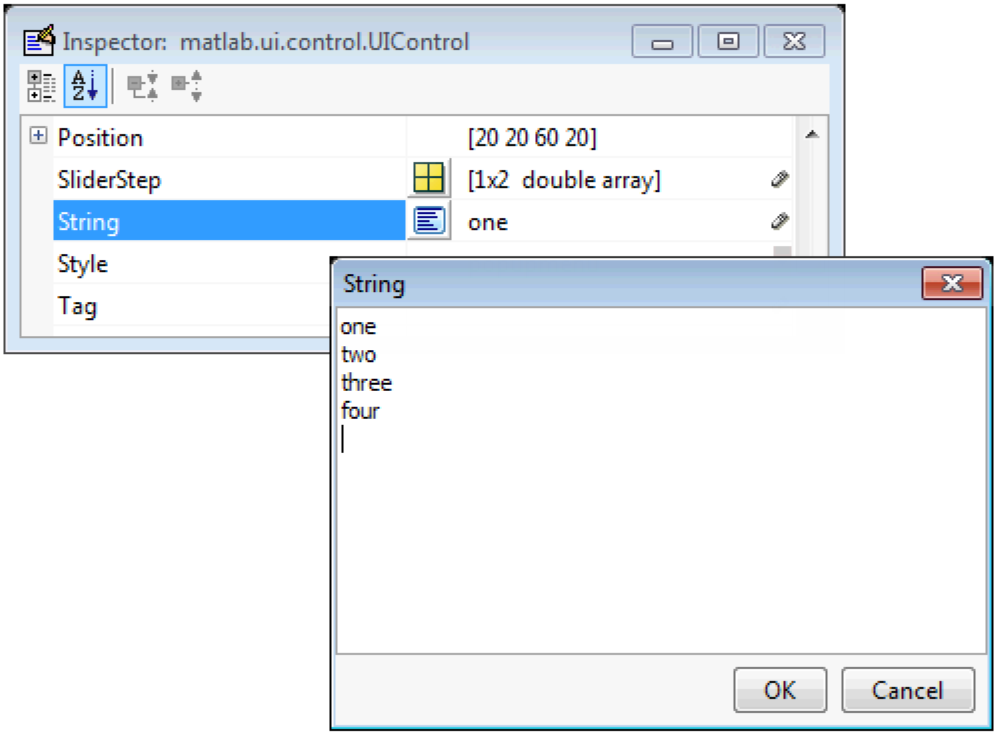
- If you want to set the position and size of the component to exact values, then modify its `Position` property. The height of a pop-up menu is determined by the font size. The height you set in the position vector is ignored.
- The pop-up menu does not let you add a label. Use a "Static Text" on page 18-16 component to label the pop-up menu.

List Box

To create a list box with items **one**, **two**, **three**, and **four**, as shown in this figure:



- Specify the list of items to be displayed by setting the `String` property to the desired list. Use the Property Inspector editor to enter the list. You can open the editor by clicking the ☰ button to the right of the property name.



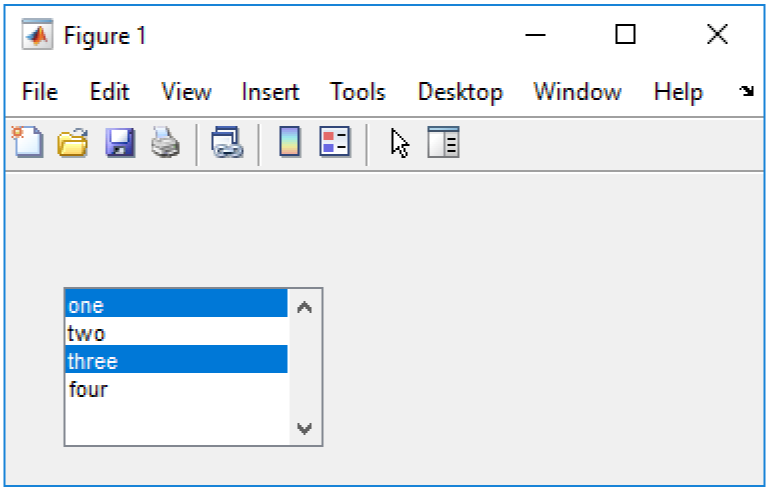
Position [20 20 60 20]
SliderStep [1x2 double array]
String one
Style
Tag

one
two
three
four

To display the & character in a label, use two & characters. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (`\`). For example, `\remove` yields **remove.**

If the width of the component is too small to accommodate one or more of the specified list items, MATLAB software truncates those items with an ellipsis.

- Specify selection by using the `Value` property together with the `Max` and `Min` properties.
  - To select a single item when the component is created, set `Value` to a scalar that indicates the index of the selected list item, where 1 corresponds to the first item in the list.
  - To select more than one item when the component is created, set `Value` to a vector of indices of the selected items. `Value = [1,3]` results in the following selection.



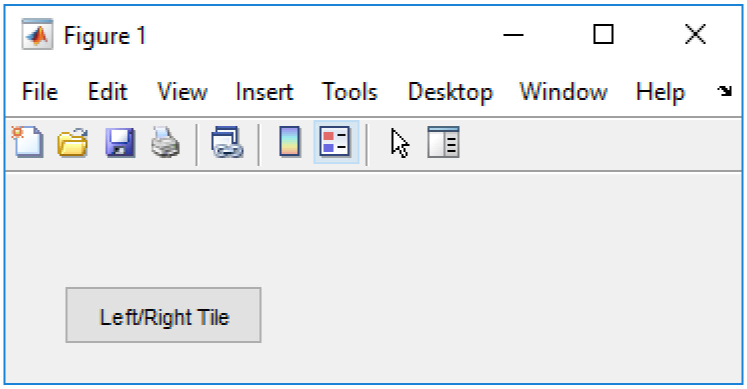
To enable selection of more than one item, you must specify the `Max` and `Min` properties so that their difference is greater than `1`. For example, `Max = 2`, `Min = 0`. `Max` default is `1`, `Min` default is `0`.

- If you want no initial selection, set the `Max` and `Min` properties to enable multiple selection, i.e., `Max - Min > 1`, and then set the `Value` property to an empty matrix `[]`.

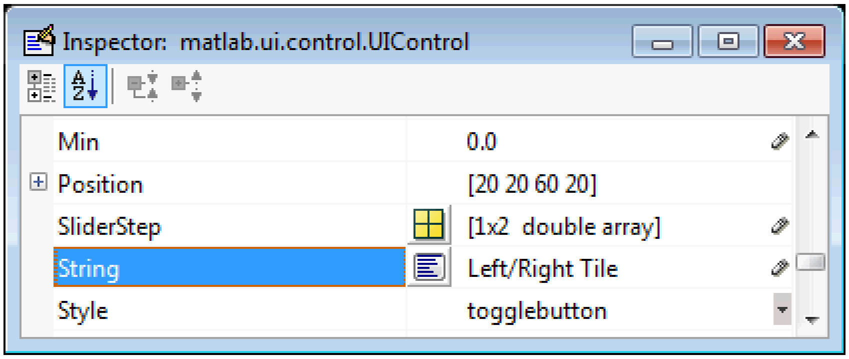
- If the list box is not large enough to display all list entries, you can set the `ListBoxTop` property to the index of the item you want to appear at the top when the component is created.
- If you want to set the position or size of the component to an exact value, then modify its `Position` property.
- The list box does not provide for a label. Use a “Static Text” on page 18-16 component to label the list box.

Toggle Button

To create a toggle button with label **Left/Right Tile**, as shown in this figure:



- Specify the toggle button label by setting its `String` property to the desired label, in this case, `Left/Right Tile`.

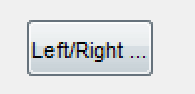


Inspector: matlab.ui.control.UIControl

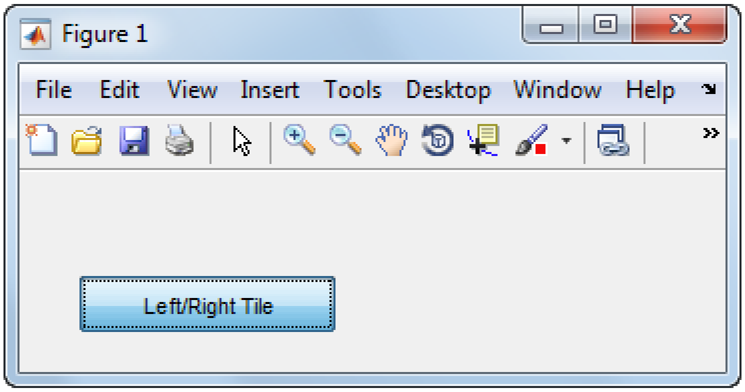
Min	0.0
⊞ Position	[20 20 60 20]
SliderStep	[1x2 double array]
String	Left/Right Tile
Style	togglebutton

To display the & character in a label, use two & characters. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (\). For example, `\remove` yields **remove.**

The toggle button accommodates only a single line of text. If you specify more than one line, only the first line is shown. If you create a toggle button that is too narrow to accommodate the specified `String` value, MATLAB truncates the text with an ellipsis.



- Create the toggle button with the button selected (depressed) by setting its `Value` property to the value of its `Max` property (default is `1`). Set `Value` to `Min` (default is `0`) to leave the toggle button unselected (raised). Correspondingly, when the user selects the toggle button, MATLAB software sets `Value` to `Max`, and to `Min` when the user deselects it. The following figure shows the toggle button in the depressed position.



- If you want to set the position or size of the component to an exact value, then modify its `Position` property.
- To add an image to a toggle button, assign the button's `CData` property an m-by-n-by-3 array of RGB values that defines a truecolor image. You must do this programmatically in the opening function of the code file. For example, the array `img` defines a 16-by-64-by-3 truecolor image using random values between 0 and 1 (generated by `rand`).

```
img = rand(16,64,3);
set(handles.togglebutton1,'CData',img);
```

where `togglebutton1` is the toggle button's `Tag` property.

To manage exclusive selection of radio buttons and toggle buttons, put them in a button group. See ButtonGroup Properties for more information.

Panels and Button Groups

Panels and button groups are containers that arrange UI components into groups. If you move the panel or button group, its children move with it and maintain their positions relative to the panel or button group.

To define panels and button groups, you must set certain properties. To do this:

1. Use the Property Inspector to modify the appropriate properties. Open the Property Inspector by selecting **View** > **Property Inspector** or by clicking the Property Inspector button .
2. In the layout area, select the component you are defining.

Subsequent topics describe commonly used properties of panels and button groups and offer a simple example for each component.

- “Commonly Used Properties” on page 18-23
- “Panel” on page 18-24
- “Button Group” on page 18-25

Commonly Used Properties

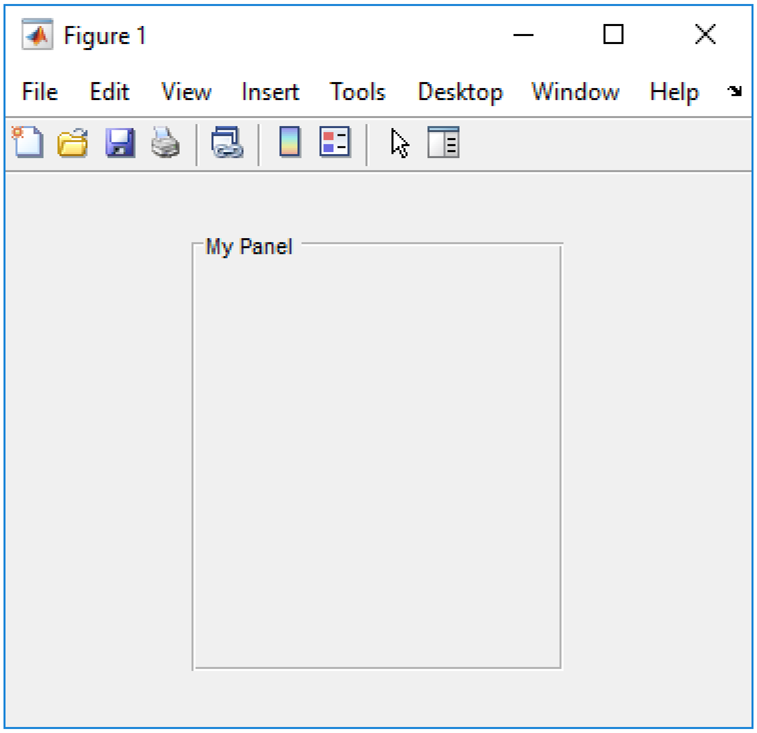
The most commonly used properties needed to describe a panel or button group are shown in the following table:

Property	Values	Description
`Position`	4-element vector: [distance from left, distance from bottom, width, height].	Size of the component and its location relative to its parent.
`Title`	Character vector (for example, `'Start'`).	Component label.
`TitlePosition`	`lefttop`, `centertop`, `righttop`, `leftbottom`, `centerbottom`, `rightbottom`. Default is `lefttop`.	Location of title in relation to the panel or button group.
`Units`	`characters`, `centimeters`, `inches`, normalized, `pixels`, `points`. Default is `characters`.	Units of measurement used to interpret the `Position` property vector

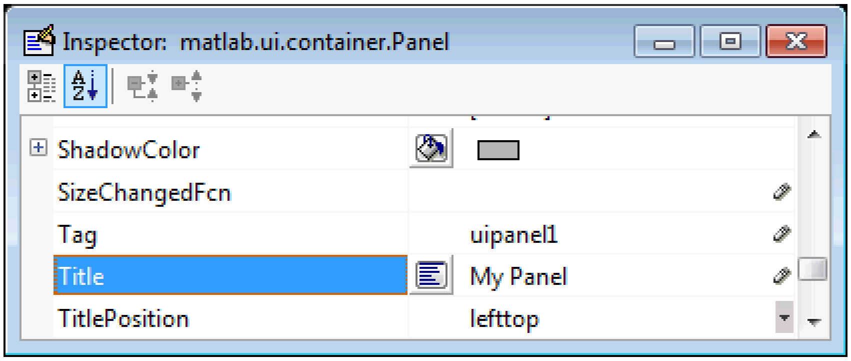
For a complete list of properties and for more information about the properties listed in the table, see the Panel Properties and ButtonGroup Properties.

Panel

To create a panel with title **My Panel** as shown in the following figure:



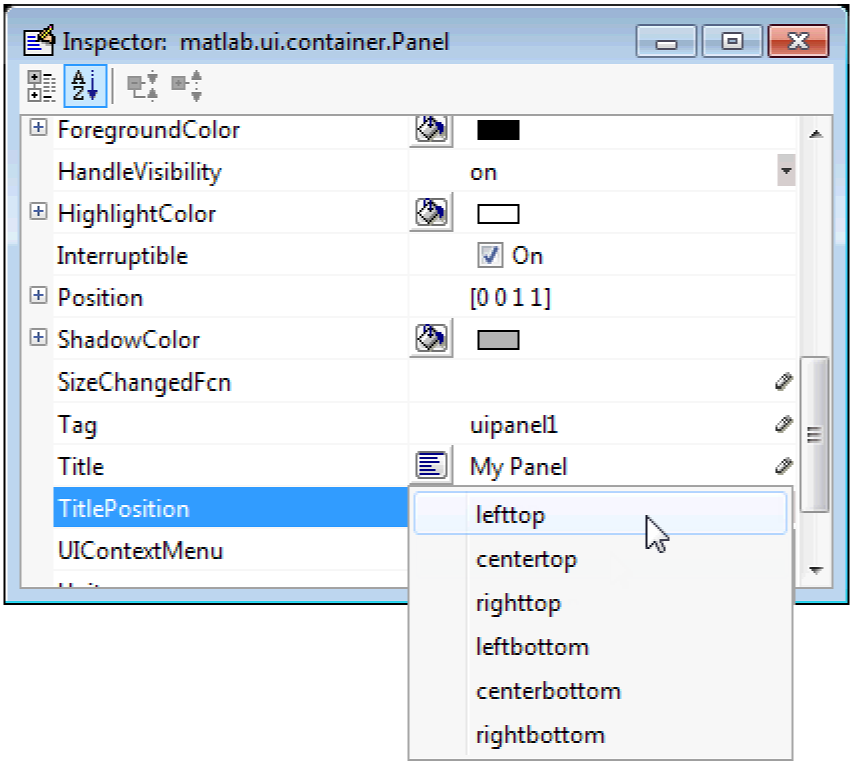
- Specify the panel title by setting the `Title` property to the desired value, in this case `My Panel`.



ShadowColor	▭
SizeChangedFcn	
Tag	uipanel1
Title	My Panel
TitlePosition	lefttop

To display the & character in the title, use two & characters. The words remove, default, and factory (case sensitive) are reserved. To use one of these as a label, prepend a backslash character (\). For example, `\remove` yields **remove**.

- Specify the location of the panel title by selecting one of the available `TitlePosition` property values from the pop-up menu, in this case `lefttop`. You can position the title at the left, middle, or right of the top or bottom of the panel.



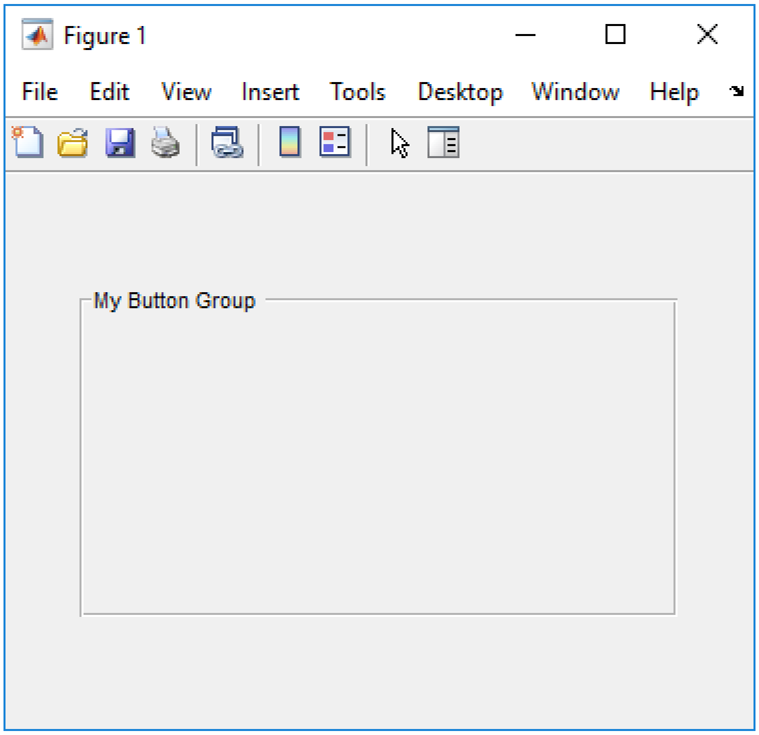
Inspector: matlab.ui.container.Panel

⊞ ForegroundColor	■
HandleVisibility	on
⊞ HighlightColor	□
Interruptible	☑ On
⊞ Position	[0 0 1 1]
⊞ ShadowColor	▨
SizeChangedFcn	
Tag	uipanel1
Title	My Panel
TitlePosition	lefttop
UIContextMenu	

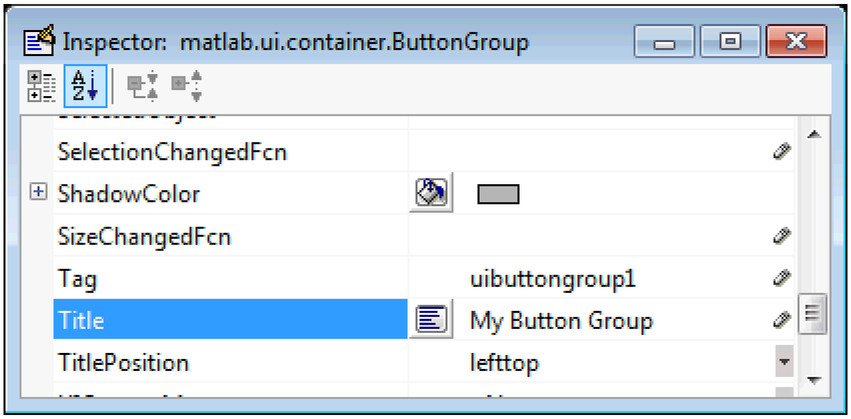
- If you want to set the position or size of the panel to an exact value, then modify its `Position` property.

Button Group

To create a button group with title **My Button Group** as shown in the following figure:



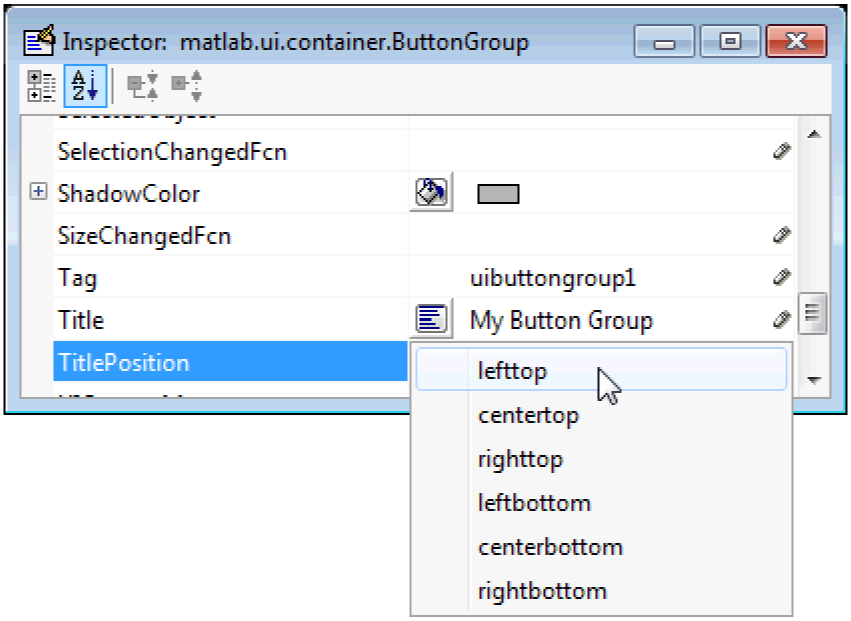
- Specify the button group title by setting the `Title` property to the desired value, in this case `My Button Group`.


Inspector: matlab.ui.container.ButtonGroup

SelectionChangedFcn	
ShadowColor	
SizeChangedFcn	
Tag	uibuttongroup1
Title	My Button Group
TitlePosition	lefttop

To display the & character in the title, use two & characters. The words remove, default, and factory (case sensitive) are reserved. To use one of these as a label, prepend a backslash characters (\). For example, `\remove` yields **remove**.

- Specify the location of the button group title by selecting one of the available `TitlePosition` property values from the pop-up menu, in this case `lefttop`. You can position the title at the left, middle, or right of the top or bottom of the button group.



SelectionChangedFcn	
ShadowColor	
SizeChangedFcn	
Tag	uibuttongroup1
Title	My Button Group
TitlePosition	

- If you want to set the position or size of the button group to an exact value, then modify its `Position` property.

Axes

Axes allow you to display graphics such as graphs and images using commands such as: `plot`, `surf`, `line`, `bar`, `pie`, `contour`, and `mesh`.

To define an axes, you must set certain properties. To do this:

1. Use the Property Inspector to modify the appropriate properties. Open the Property Inspector by selecting **View** > **Property Inspector** or by clicking the Property Inspector button.
2. In the layout area, select the component you are defining.

Subsequent topics describe commonly used properties of axes and offer a simple example.

- “Commonly Used Properties” on page 18-27
- “Create Axes” on page 18-28

Commonly Used Properties

The most commonly used properties needed to describe an axes are shown in the following table:

Property	Values	Description
`NextPlot`	`add`, `replace`, `replacechildren`. Default is `replace`	Specifies whether plotting adds graphics, replaces graphics and resets axes properties to default, or replaces graphics only.

Property	Values	Description
Position	4-element vector: [distance from left, distance from bottom, width, height].	Size of the component and its location relative to its parent.
Units	`normalized`, `centimeters`, `characters`, `inches`, `pixels`, `points`. Default is `normalized`.	Units of measurement used to interpret position vector

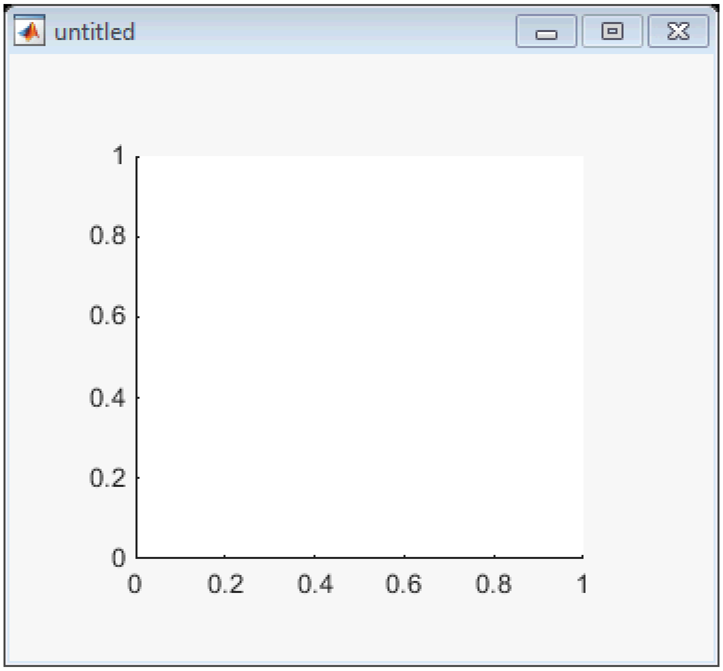
For a complete list of properties and for more information about the properties listed in the table, see Axes.

See commands such as the following for more information on axes objects: `plot`, `surf`, `line`, `bar`, `polar`, `pie`, `contour`, `imagesc`, and `mesh`.

Many of these graphing functions reset axes properties by default, according to the setting of its `NextPlot` property, which can cause unwanted behavior, such as resetting axis limits and removing axes context menus and callbacks. See “Create Axes” on page 18-28 for information about setting the `NextPlot` property.

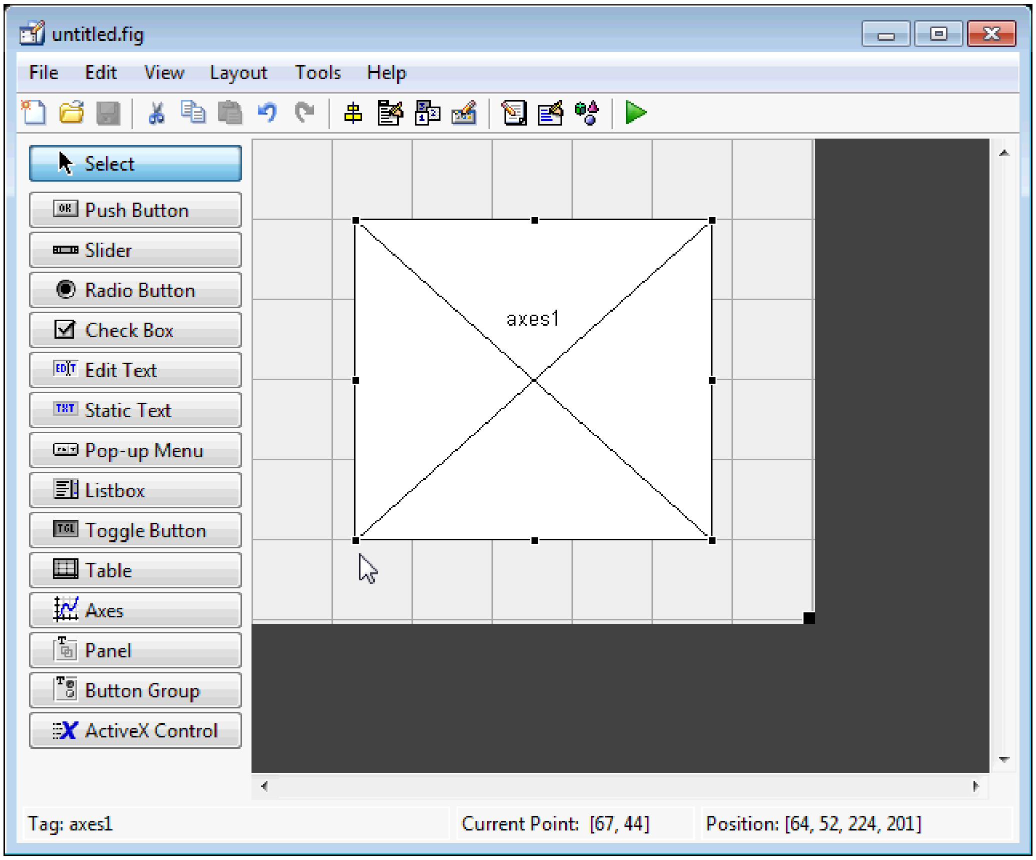
Create Axes

Here is an axes in a GUIDE app:



Use these guidelines when you create axes objects in GUIDE:

- Allow for tick marks to be placed outside the box that appears in the Layout Editor. The axes above looks like this in the layout editor; placement allows space at the left and bottom of the axes for tick marks. Functions that draw in the axes update the tick marks appropriately.



Tag: axes1

Current Point: [67, 44]

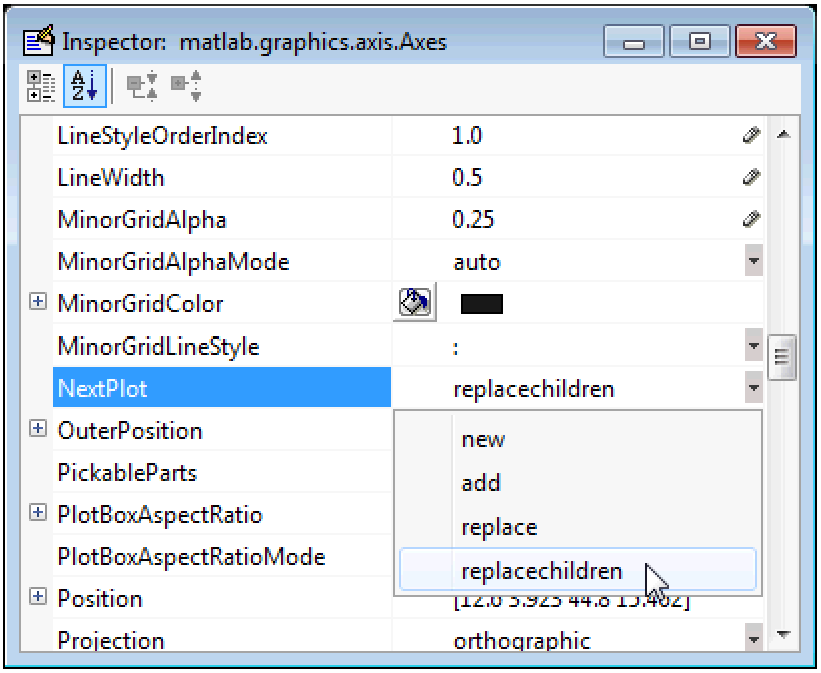
Position: [64, 52, 224, 201]

- Use the `title`, `xlabel`, `ylabel`, `zlabel`, and `text` functions in the code file to label an axes component. For example,

  ```
  xlh = (axes_handle,'Years')
  ```

  labels the X-axis as `Years`. The handle of the X-axis label is `xlh`.

  The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these in component text, prepend a backslash character (`\`). For example, `\remove` yields **remove.**
- If you want to set the position or size of the axes to an exact value, then modify its `Position` property.
- If you customize axes properties, some of them (or example, callbacks, font characteristics, and axis limits and ticks) may get reset to default every time you draw a graph into the axes when the `NextPlot` property has its default value of `'replace'`. To keep customized properties as you want them, set `NextPlot` to `'replacechildren'` in the Property Inspector, as shown here.



Inspector: matlab.graphics.axis.Axes

LineStyleOrderIndex	1.0
LineWidth	0.5
MinorGridAlpha	0.25
MinorGridAlphaMode	auto
MinorGridColor	■
MinorGridLineStyle	:
NextPlot	replacechildren
OuterPosition	
PickableParts	
PlotBoxAspectRatio	
PlotBoxAspectRatioMode	
Position	[12.6 3.923 44.8 15.462]
Projection	orthographic

Table

Tables enable you to display data in a two dimensional table. You can use the Property Inspector to get and set the object property values.

Commonly Used Properties

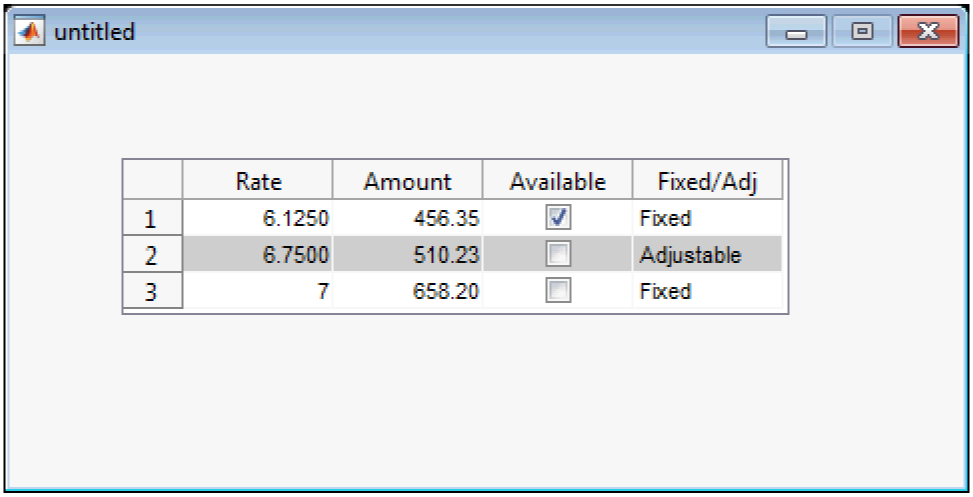
The most commonly used properties of a table component are listed in the table below. These are grouped in the order they appear in the Table Property Editor. Please refer to `uitable` documentation for detail of all the table properties:

Group	Property	Values	Description
Column	`ColumnName`	1-by-*n* cell array of character vectors | {'numbered'} | empty matrix ([])	The header label of the column.
	`ColumnFormat`	Cell array of character vectors	Determines display and editability of columns
	`ColumnWidth`	1-by-*n* cell array or `'auto'`	Width of each column in pixels; individual column widths can also be set to `'auto'`
	`ColumnEditable`	logical 1-by-*n* matrix | scalar logical value | empty matrix ([])	Determines data in a column as editable

Group	Property	Values	Description
Row	`RowName`	1-by-*n* cell array of character vectors	Row header label names
Color	`BackgroundColor`	*n*-by-3 matrix of RGB triples	Background color of cells
	`RowStriping`	{on} | off	Color striping of table rows
Data	`Data`	Matrix or cell array of numeric, logical, or character data	Table data.

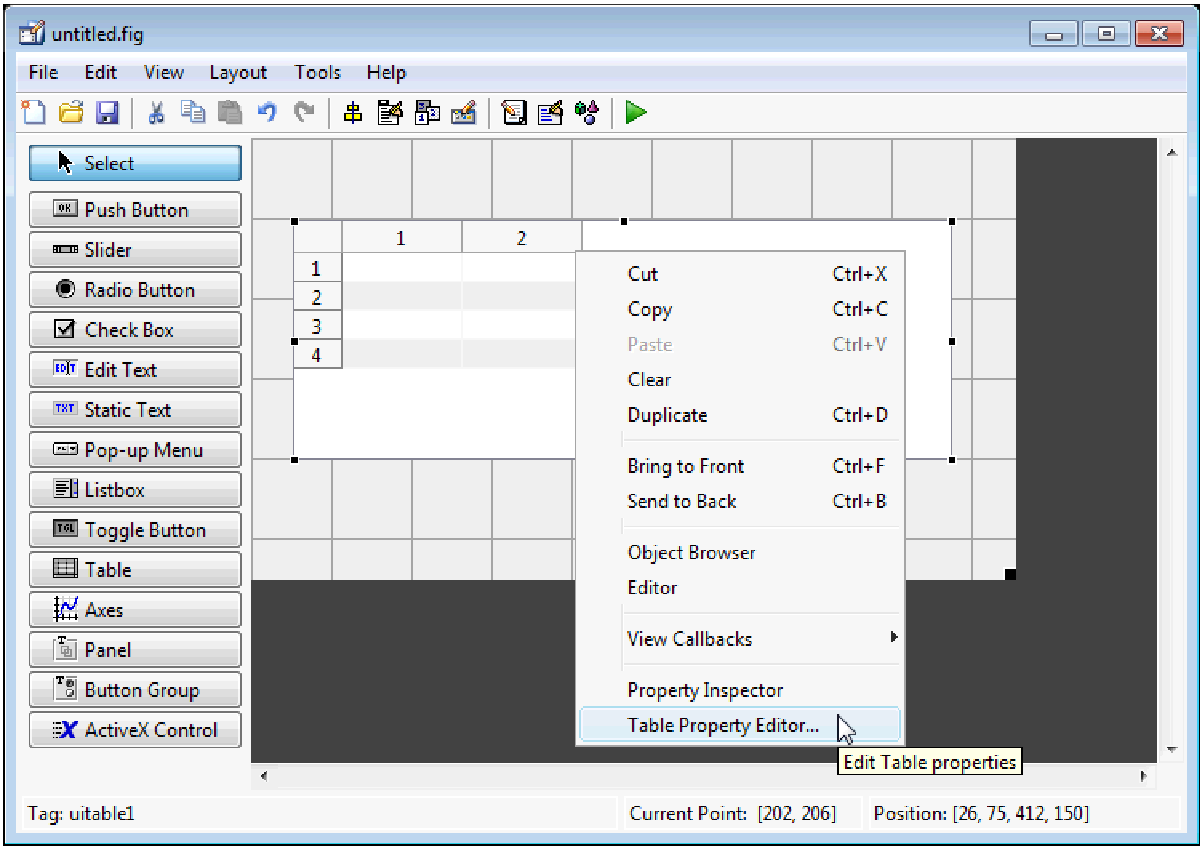
Create a Table

To create a UI with a table in GUIDE as shown, do the following:



	Rate	Amount	Available	Fixed/Adj
1	6.1250	456.35	☑	Fixed
2	6.7500	510.23	□	Adjustable
3	7	658.20	□	Fixed

Drag the table icon on to the Layout Editor and right click in the table. From the table’s context menu, select **Table Property Editor**. You can also select **Table Property Editor** from the **Tools** menu when you select a table by itself.



Tag: uitable1 Current Point: [202, 206] Position: [26, 75, 412, 150]

Use the Table Property Editor

When you open it this way, the Table Property Editor displays the **Column** pane. You can also open it from the Property Inspector by clicking one of its Table Property Editor icons, in which case the Table Property Editor opens to display the pane appropriate for the property you clicked.

Clicking items in the list on the left hand side of the Table Property Editor changes the contents of the pane to the right . Use the items to activate controls for specifying the table's **Columns**, **Rows**, **Data**, and **Color** options.

The **Columns** and **Rows** panes each have a data entry area where you can type names and set properties on a per-column or per-row basis. You can edit only one row or column definition at a time. These panes contain a vertical group of five buttons for editing and navigating:

Button	Purpose	Accelerator Keys	
		Windows	Macintosh
Insert	Inserts a new column or row definition entry below the current one	**Insert**	**Insert**
Delete	Deletes the current column or row definition entry (no undo)	**Ctrl+D**	**Cmd+D**
Copy	Inserts a Copy of the selected entry in a new row below it	**Ctrl+P**	**Cmd+P**

Button	Purpose	Accelerator Keys	
		Windows	Macintosh
Up	Moves selected entry up one row	**Ctrl+** **Up Arrow**	**Cmd+** **Up Arrow**
Down	Moves selected entry down one row	**Ctrl+** **Down Arrow**	**Cmd+** **Down Arrow**

Keyboard equivalents only operate when the cursor is in the data entry area. In addition to those listed above, typing **Ctrl+T** or **Cmd+T** selects the entire field containing the cursor for editing (if the field contains text).

To save changes to the table you make in the Table Property Editor, click **OK**, or click **Apply** commit changes and keep on using the Table Property Editor.

Set Column Properties

Click **Insert** to add two more columns.

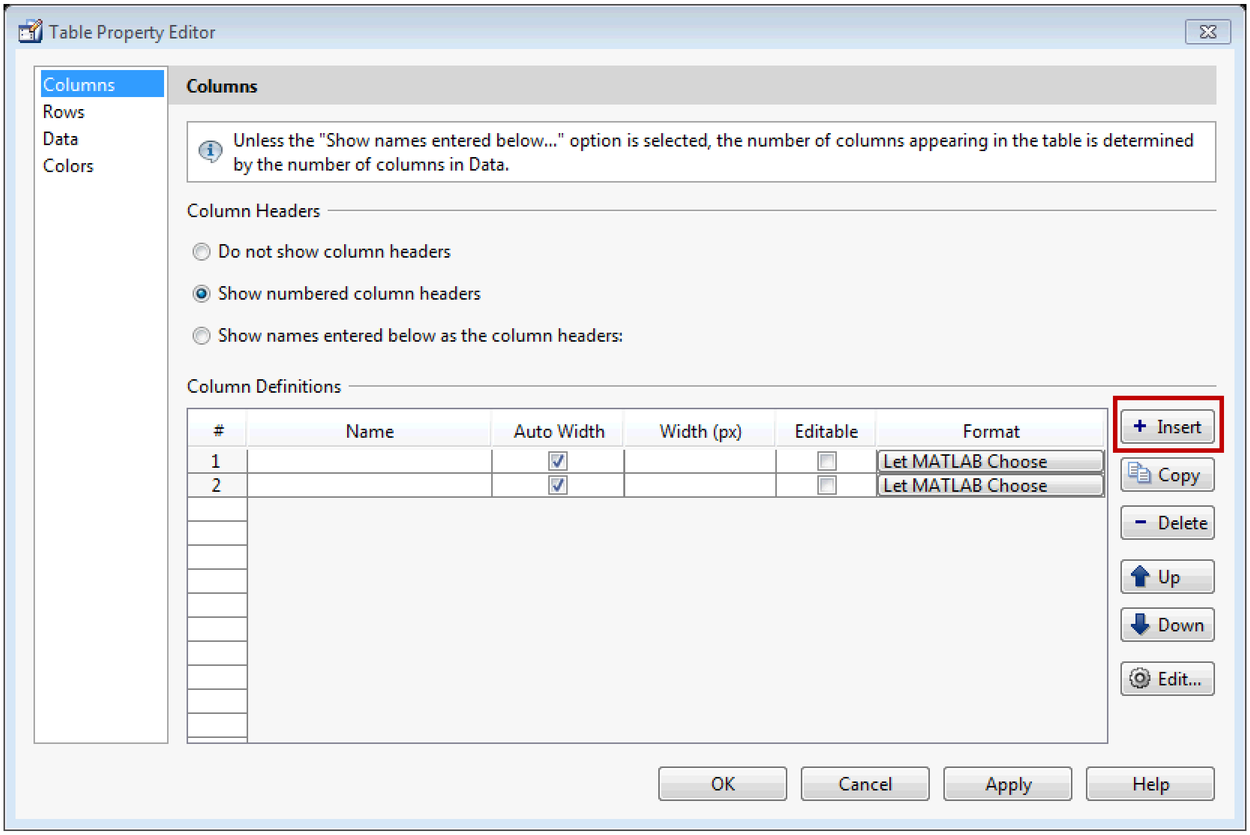


Table Property Editor

Columns

Unless the "Show names entered below..." option is selected, the number of columns appearing in the table is determined by the number of columns in Data.

Column Headers

○ Do not show column headers

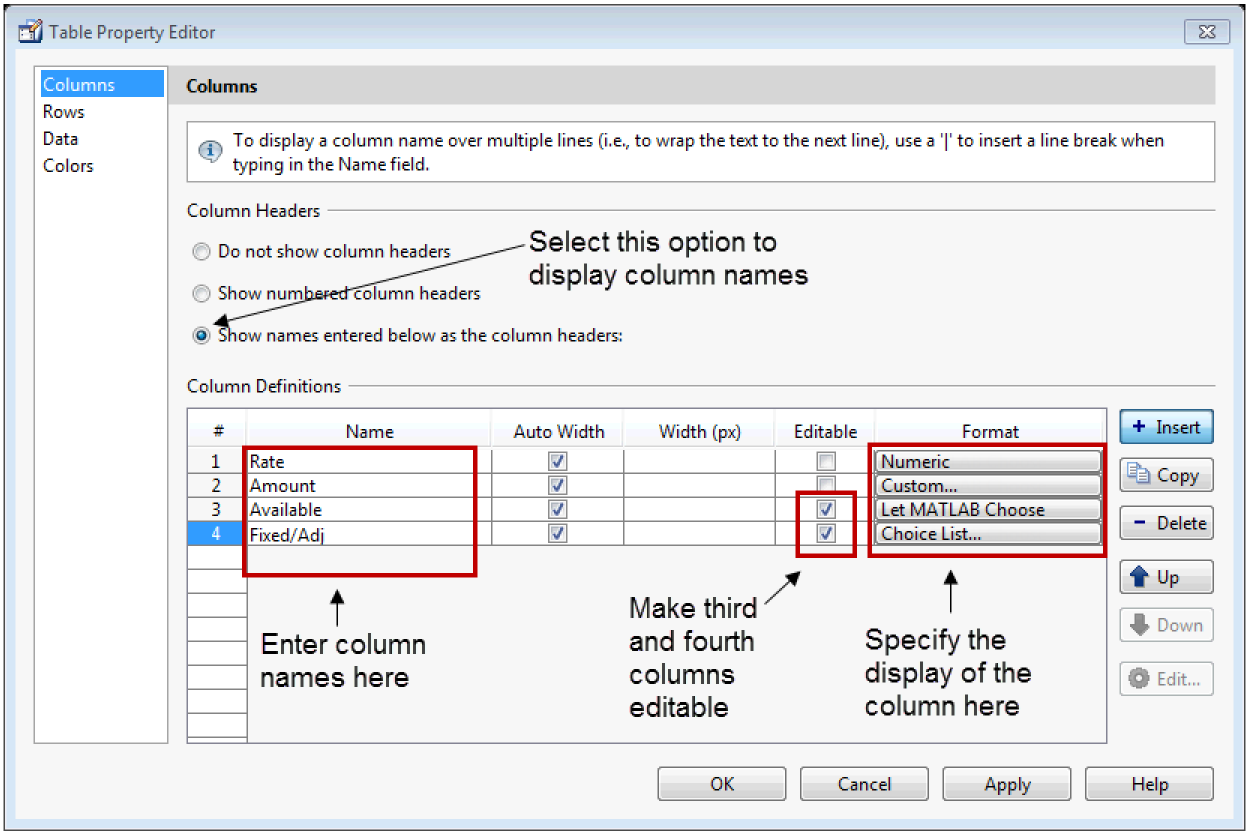
● Show numbered column headers

○ Show names entered below as the column headers:

Column Definitions

#	Name	Auto Width	Width (px)	Editable	Format
1		☑		□	Let MATLAB Choose
2		☑		□	Let MATLAB Choose

Select **Show names entered below as the column headers** and set the `ColumnName` by entering Rate, Amount, Available, and Fixed/Adj in **Name** group. for the Available and Fixed/Adj columns set the `ColumnEditable` property to `on`. Lastly set the `ColumnFormat` for the four columns.



Columns

To display a column name over multiple lines (i.e., to wrap the text to the next line), use a '|' to insert a line break when typing in the Name field.

Column Headers

○ Do not show column headers

○ Show numbered column headers

● Show names entered below as the column headers:

Column Definitions

#	Name	Auto Width	Width (px)	Editable	Format
1	Rate	☑		☐	Numeric
2	Amount	☑		☐	Custom...
3	Available	☑		☑	Let MATLAB Choose
4	Fixed/Adj	☑		☑	Choice List...

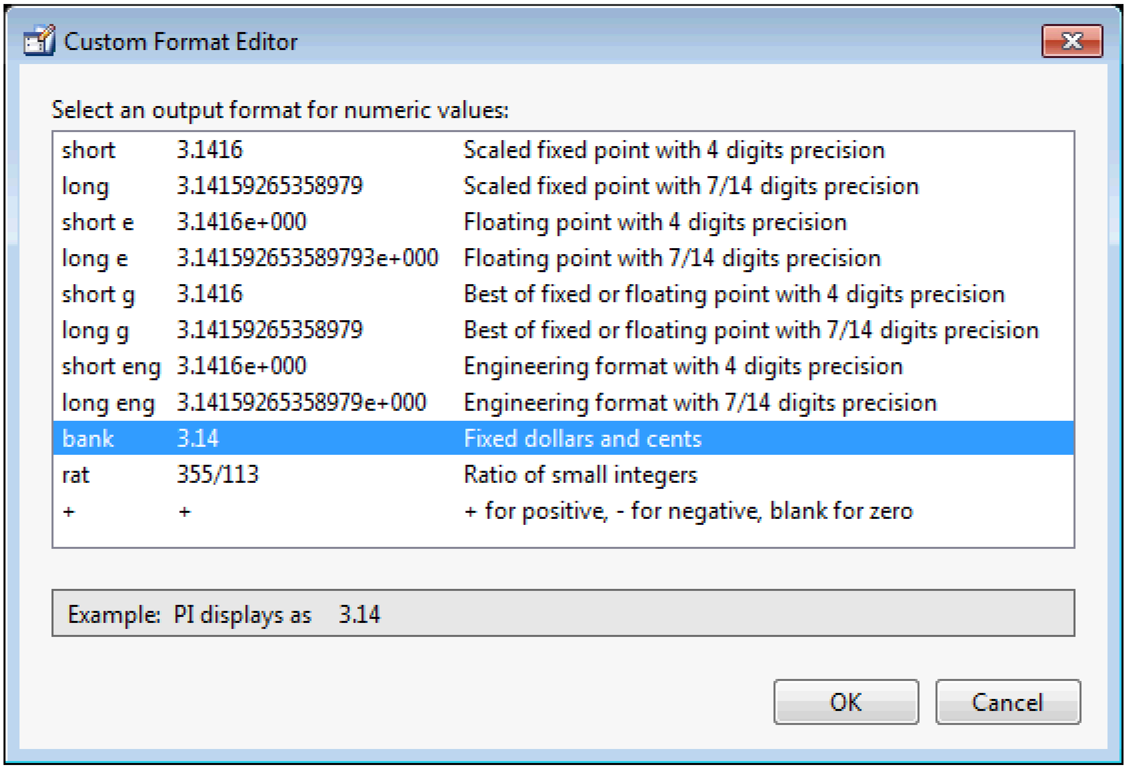
Select this option to display column names

Enter column names here

Make third and fourth columns editable

Specify the display of the column here

For the Rate column, select **Numeric**. For the Amount Column select **Custom** and in the Custom Format Editor, choose **Bank**.

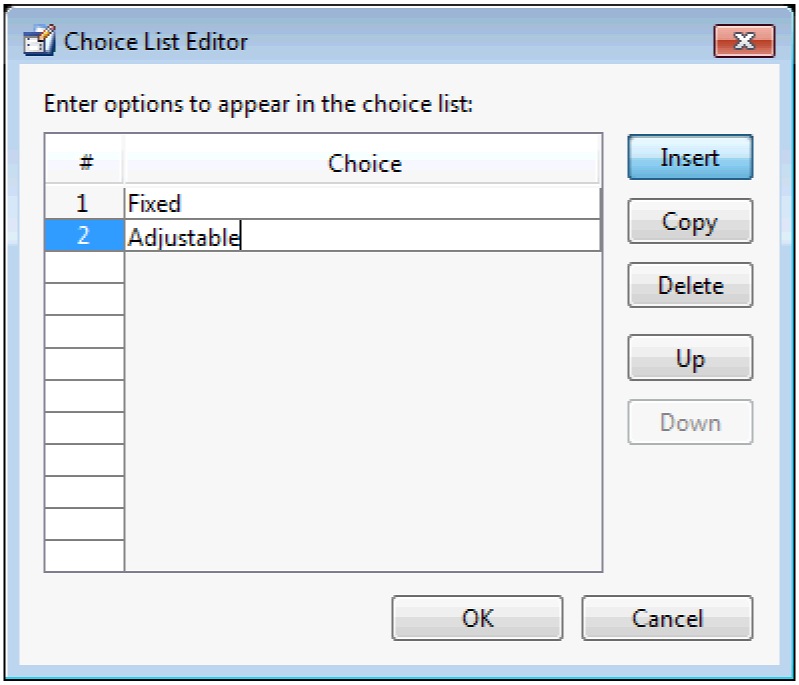


Select an output format for numeric values:

short	3.1416	Scaled fixed point with 4 digits precision
long	3.14159265358979	Scaled fixed point with 7/14 digits precision
short e	3.1416e+000	Floating point with 4 digits precision
long e	3.141592653589793e+000	Floating point with 7/14 digits precision
short g	3.1416	Best of fixed or floating point with 4 digits precision
long g	3.14159265358979	Best of fixed or floating point with 7/14 digits precision
short eng	3.1416e+000	Engineering format with 4 digits precision
long eng	3.14159265358979e+000	Engineering format with 7/14 digits precision
bank	3.14	Fixed dollars and cents
rat	355/113	Ratio of small integers
+	+	+ for positive, - for negative, blank for zero

Example: PI displays as 3.14

Leave the Available column at the default value. This allows MATLAB to chose based on the value of the `Data` property of the table. For the Fixed/Adj column select `Choice List` to create a pop-up menu. In the Choice List Editor, click **Insert** to add a second choice and type Fixed and Adjustable as the 2 choices.



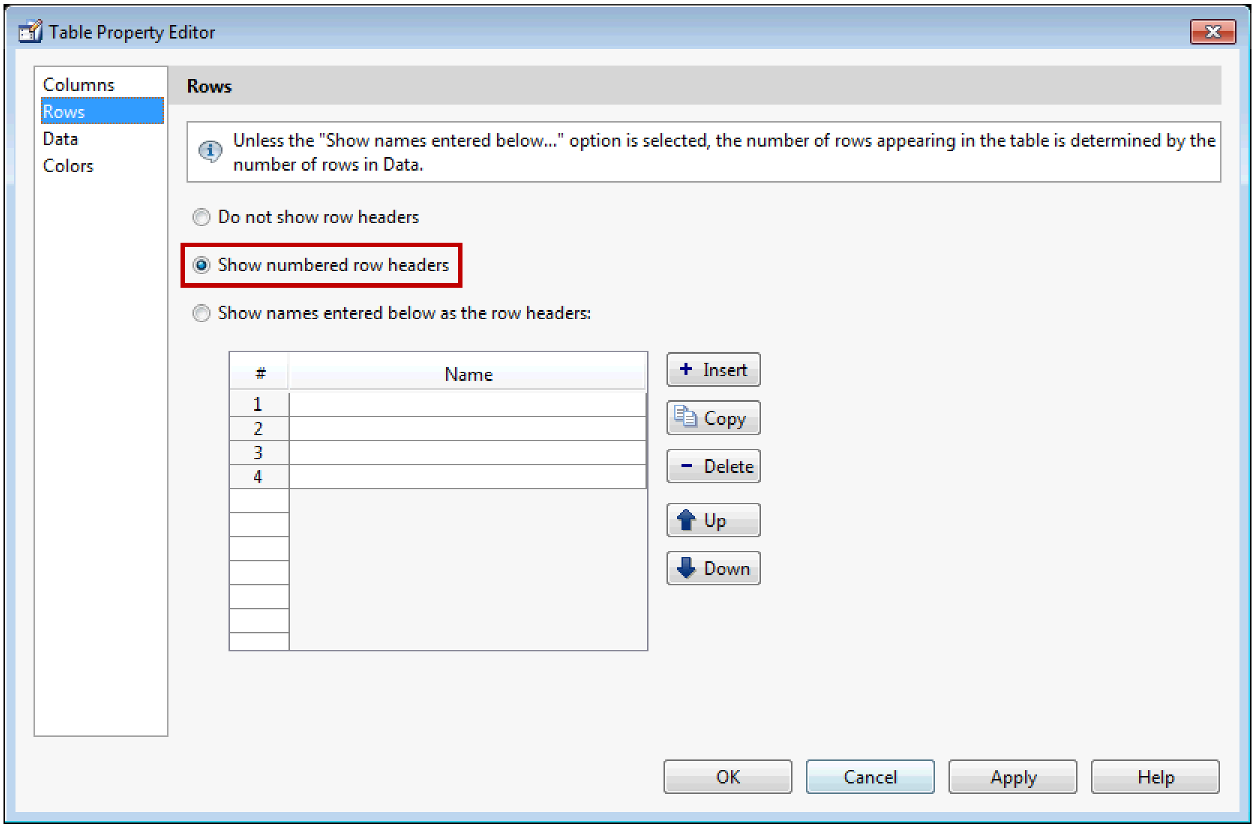
Enter options to appear in the choice list:

#	Choice
1	Fixed
2	Adjustable

Note For a user to select items from a choice list, the `ColumnEditable` property of the column that the list occupies must be set to `'true'`. The pop-up control only appears when the column is editable.

Set Row Properties

In the Row tab, leave the default `RowName`, **Show numbered row headers**.



Unless the "Show names entered below..." option is selected, the number of rows appearing in the table is determined by the number of rows in Data.

○ Do not show row headers

● Show numbered row headers

○ Show names entered below as the row headers:

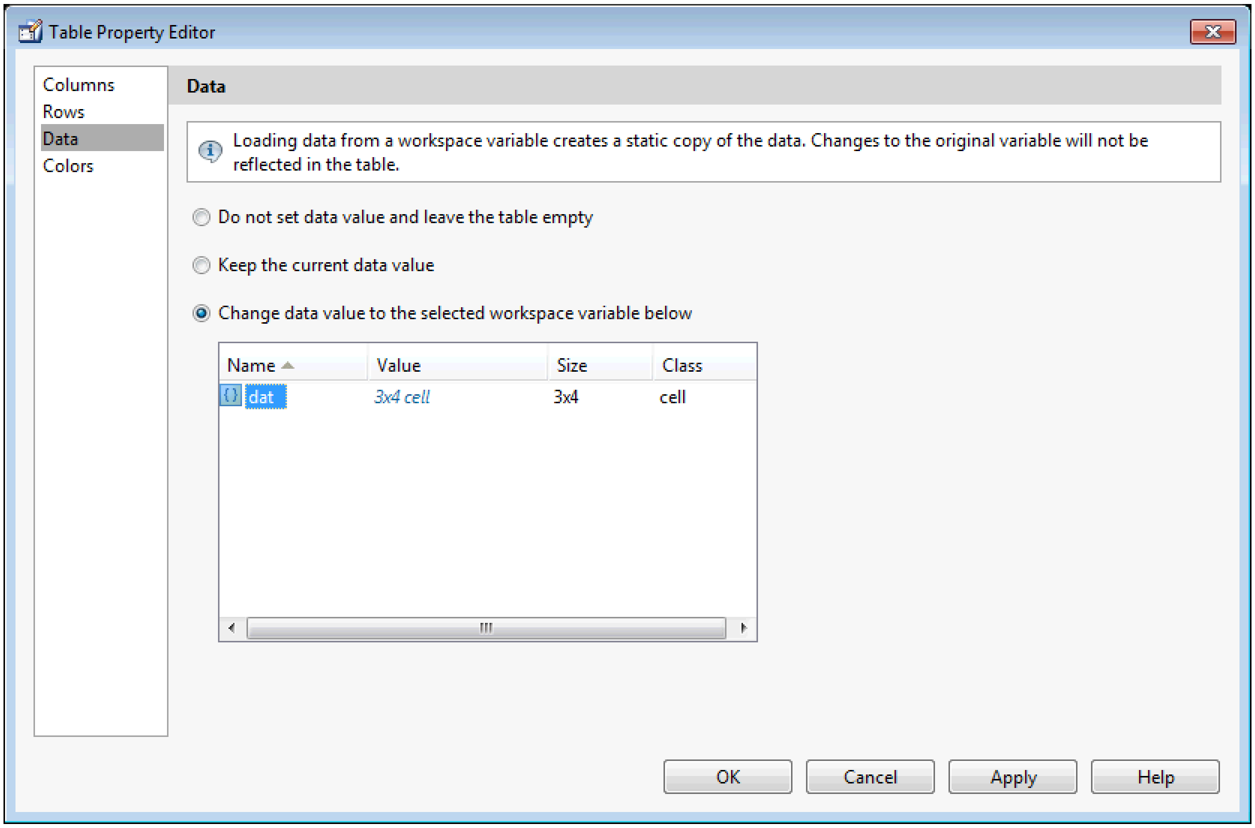
#	Name
1	
2	
3	
4	

Set Data Properties

Use the `Data` property to specify the data in the table. Create the data in the command window before you specify it in GUIDE. For this example, type:

```
dat =  {6.125, 456.3457, true,  'Fixed';...
6.75,  510.2342, false, 'Adjustable';...
7,     658.2,    false, 'Fixed';};
```

In the Table Property Editor, select the data that you defined and select **Change data value to the selected workspace variable below**.



Loading data from a workspace variable creates a static copy of the data. Changes to the original variable will not be reflected in the table.

○ Do not set data value and leave the table empty

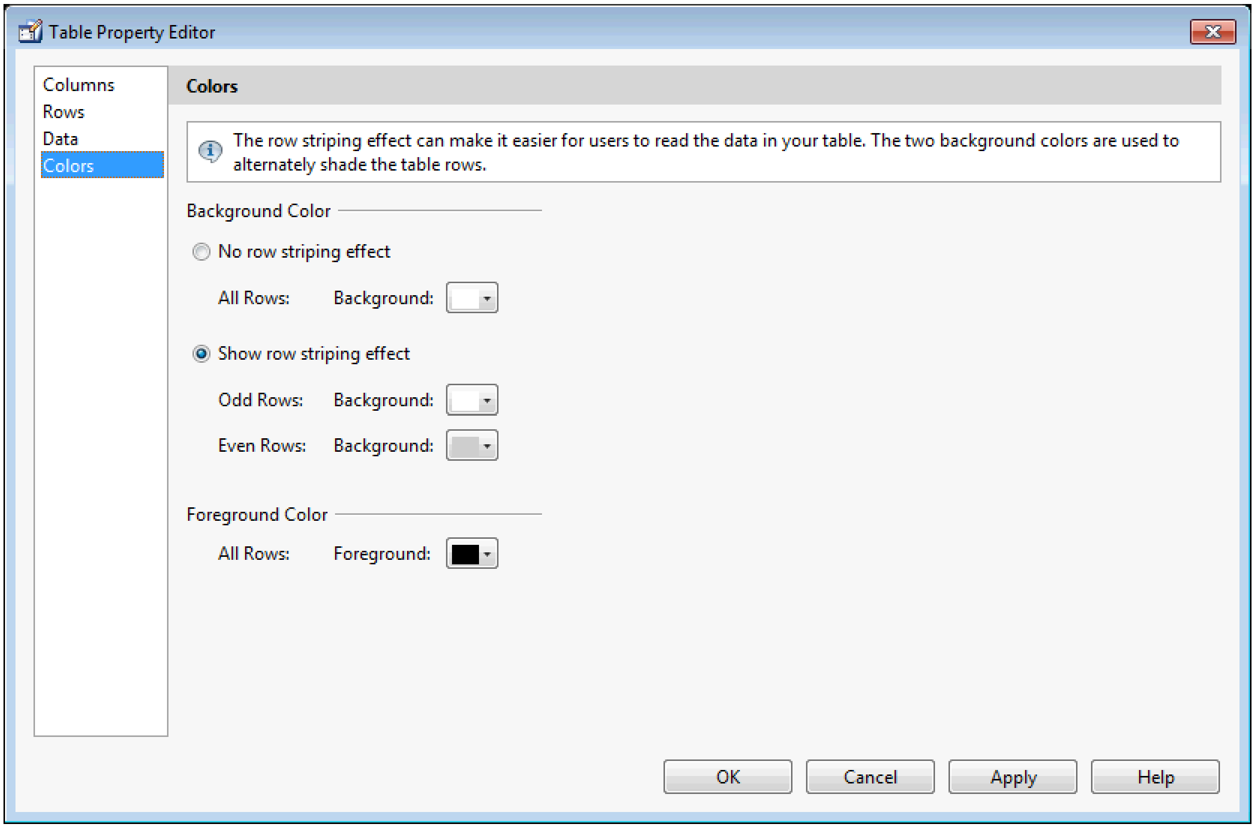
○ Keep the current data value

● Change data value to the selected workspace variable below

Name	Value	Size	Class
dat	3x4 cell	3x4	cell

Set Color Properties

Specify the `BackgroundColor` and `RowStriping` for your table in the Color tab.



Colors

The row striping effect can make it easier for users to read the data in your table. The two background colors are used to alternately shade the table rows.

Background Color

○ No row striping effect

All Rows: Background:

● Show row striping effect

Odd Rows: Background:

Even Rows: Background:

Foreground Color

All Rows: Foreground:

You can change other `uitable` properties to the table via the Property Inspector.

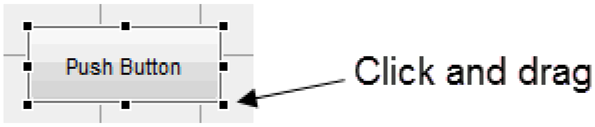
Resize GUIDE UI Components

You can resize components in one of the following ways:

- "Drag a Corner of the Component" on page 18-38
- "Set the Component's Position Property" on page 18-38

Drag a Corner of the Component

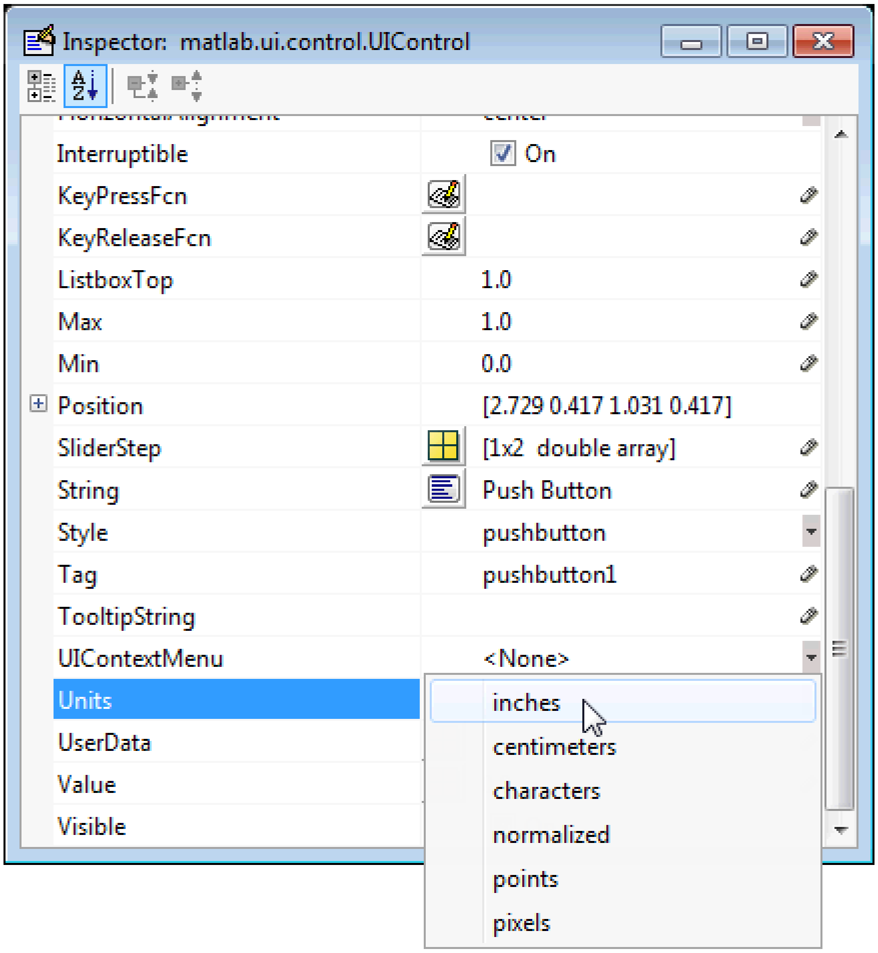
Select the component you want to resize. Click one of the corner handles and drag it until the component is the desired size.



Set the Component's Position Property

Select one or more components that you want to resize. Then select **View** > **Property Inspector**or click the Property Inspector button 📝.

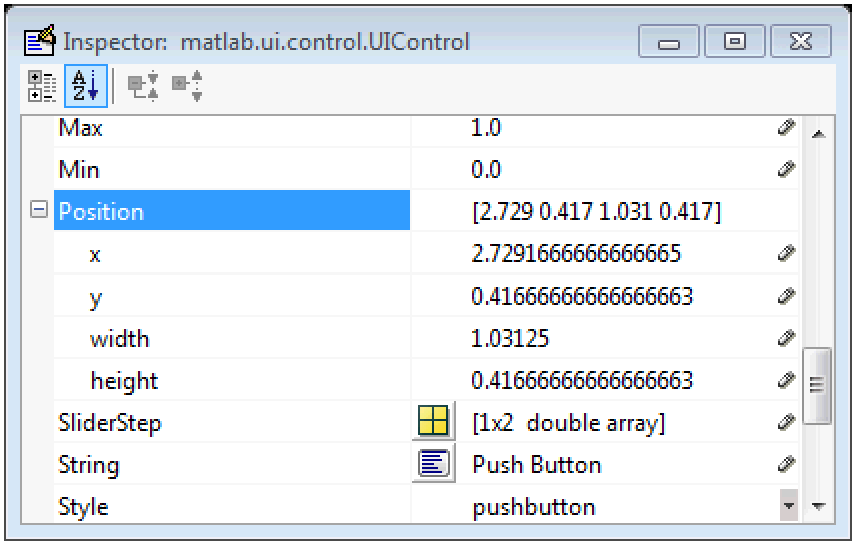
1 In the Property Inspector, scroll to the `Units` property and note whether the current setting is `characters` or `normalized`. Click the button next to `Units` and then change the setting to `inches` from the pop-up menu.



Inspector: matlab.ui.control.UIControl

Interruptible	☑ On
KeyPressFcn	
KeyReleaseFcn	
ListboxTop	1.0
Max	1.0
Min	0.0
Position	[2.729 0.417 1.031 0.417]
SliderStep	[1x2 double array]
String	Push Button
Style	pushbutton
Tag	pushbutton1
TooltipString	
UIContextMenu	<None>
Units	
UserData	
Value	
Visible	

2 Click the + sign next to `Position`. The Property Inspector displays the elements of the `Position` property.



Inspector: matlab.ui.control.UIControl

Max	1.0
Min	0.0
Position	[2.729 0.417 1.031 0.417]
x	2.7291666666666665
y	0.41666666666666663
width	1.03125
height	0.41666666666666663
SliderStep	[1x2 double array]
String	Push Button
Style	pushbutton

3 Type the `width` and `height` you want the components to be.

4 Reset the `Units` property to its previous setting, either `characters` or `normalized`.

To select multiple components, they must have the same parent. That is, they must be contained in the same figure, panel, or button group. Setting the `Units` property to `characters` (nonresizable UIs) or `normalized` (resizable UIs) gives the UI a more consistent appearance across platforms.

See Also

Related Examples

- “Ways to Build Apps” on page 1-2
- “Write Callbacks in GUIDE” on page 19-2
- “Callbacks for Specific Components” on page 19-14
- “Lay Out Apps in App Designer Design View” on page 5-2
- “App Building Components” on page 4-2

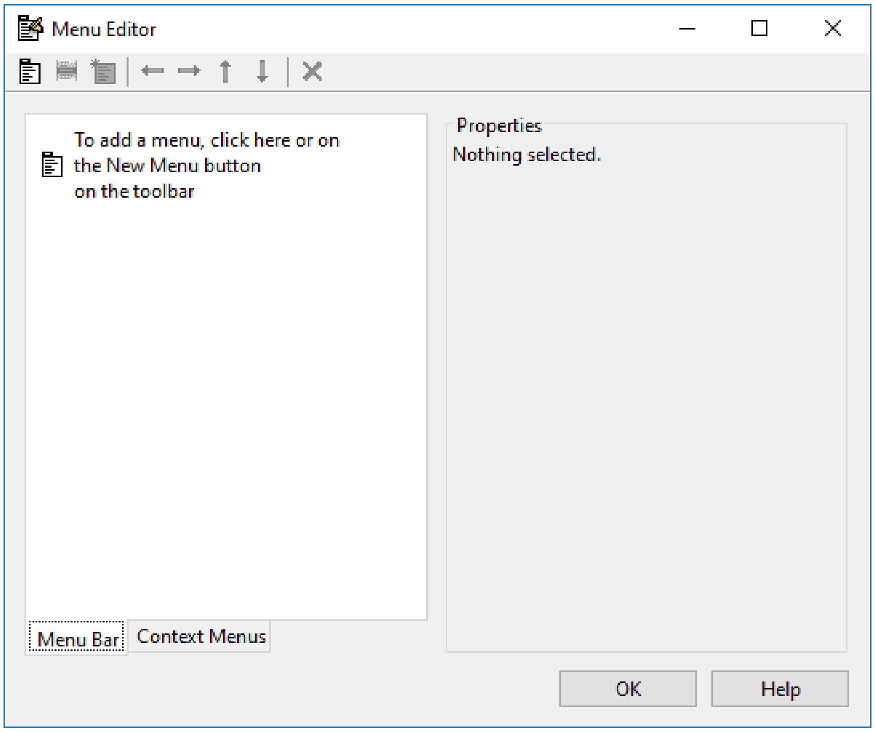
Create Menus for GUIDE Apps

In this section...
"Menus for the Menu Bar" on page 18-41
"Context Menus" on page 18-48

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see "GUIDE Migration Strategies" on page 3-7 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, "Develop Apps Using App Designer" instead.

You can use GUIDE to create menu bars (containing pull-down menus) as well as context menus that you attach to components. You can create both types of menus using the Menu Editor. Access the Menu Editor from the **Tools** menu or click the **Menu Editor** button.


Menu Editor
To add a menu, click here or on the New Menu button on the toolbar
Properties
Nothing selected.
Menu Bar
Context Menus
OK
Help

Menus for the Menu Bar

- "How Menus Affect Figure Docking" on page 18-42
- "Add Standard Menus to the Menu Bar" on page 18-43
- "Create a Menu" on page 18-43

- “Add Items to a Menu” on page 18-44
- “Additional Drop-Down Menus” on page 18-46
- “Cascading Menus” on page 18-46

When you create a drop-down menu, GUIDE adds its title to the menu bar. You then can create menu items for that menu. Each menu item can have a cascading menu, also known as a submenu, and these items can have cascading menus, and so on.

How Menus Affect Figure Docking

By default, when you create a UI with GUIDE, it does not create a menu bar for that UI. You might not need menus for your UI, but if you want the user to be able to dock or undock the UI window, it must contain a menu bar or a toolbar. This is because docking is controlled by the docking icon, a small curved arrow near the upper-right corner of the menu bar or the toolbar, as the following illustration shows.

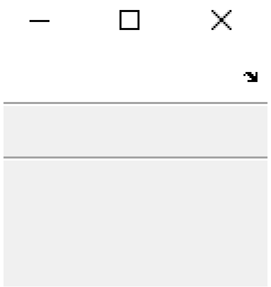


Figure windows with a standard menu bar also have a **Desktop** menu from which the user can dock and undock them.

To display the docking arrow and the **Desktop** > **Dock Figure** menu item, use the Property Inspector to set the figure property `DockControls` to `'on'`. You must also set the `MenuBar` and/or `ToolBar` figure properties to `'figure'` to display docking controls.

The `WindowStyle` figure property also affects docking behavior. The default is `'normal'`, but if you change it to `'docked'`, then the following applies:

- The UI window opens docked in the desktop when you run it.
- The `DockControls` property is set to `'on'` and cannot be turned off until `WindowStyle` is no longer set to `'docked'`.
- If you undock a UI window created with `WindowStyle 'docked'`, it will have not have a docking arrow unless the figure displays a menu bar or a toolbar (either standard or customized). When it has no docking arrow, users can undock it from the desktop, but will be unable to redock it there.

However, when you provide your own menu bar or toolbar using GUIDE, it can display the docking arrow if you want the UI window to be dockable.

Note UIs that are modal dialogs (figures with `WindowStyle` set to `'modal'`) cannot have menu bars, toolbars, or docking controls.

For more information, see the `DockControls`, `MenuBar`, `ToolBar`, and `WindowStyle` property descriptions in Figure.

Add Standard Menus to the Menu Bar

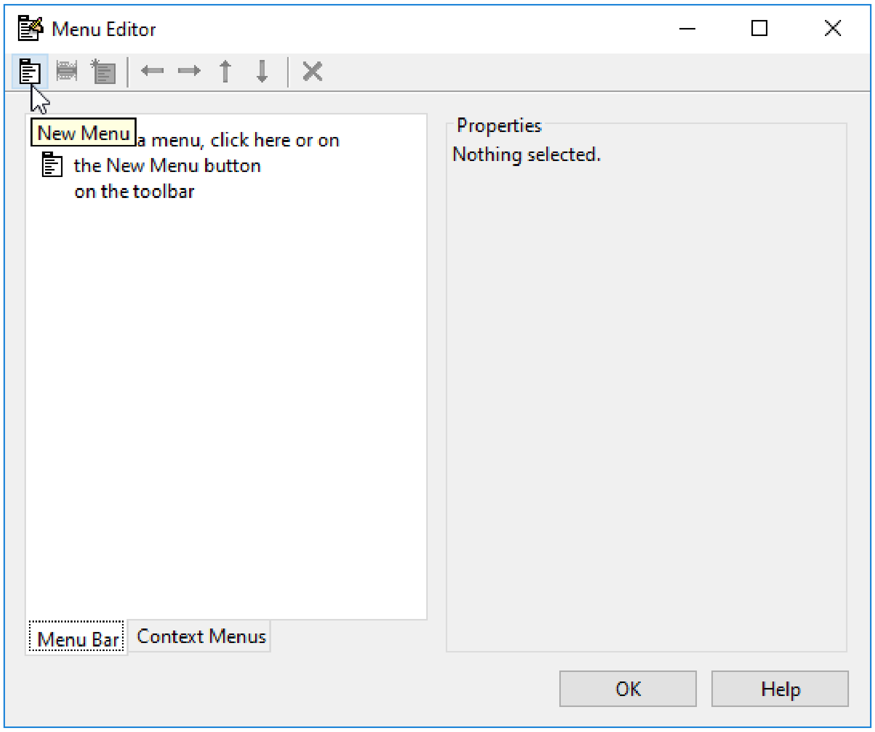
The figure `MenuBar` property controls whether your UI displays the MATLAB standard menus on the menu bar. GUIDE initially sets the value of `MenuBar` to `none`. If you want your UI to display the MATLAB standard menus, use the Property Inspector to set `MenuBar` to `figure`.

- If the value of `MenuBar` is `none`, GUIDE automatically adds a menu bar that displays only the menus you create.
- If the value of `MenuBar` is `figure`, the UI displays the MATLAB standard menus and GUIDE adds the menus you create to the right side of the menu bar.

In either case, you can enable the user to dock and undock the window by setting the figure's `DockControls` property to `'on'`.

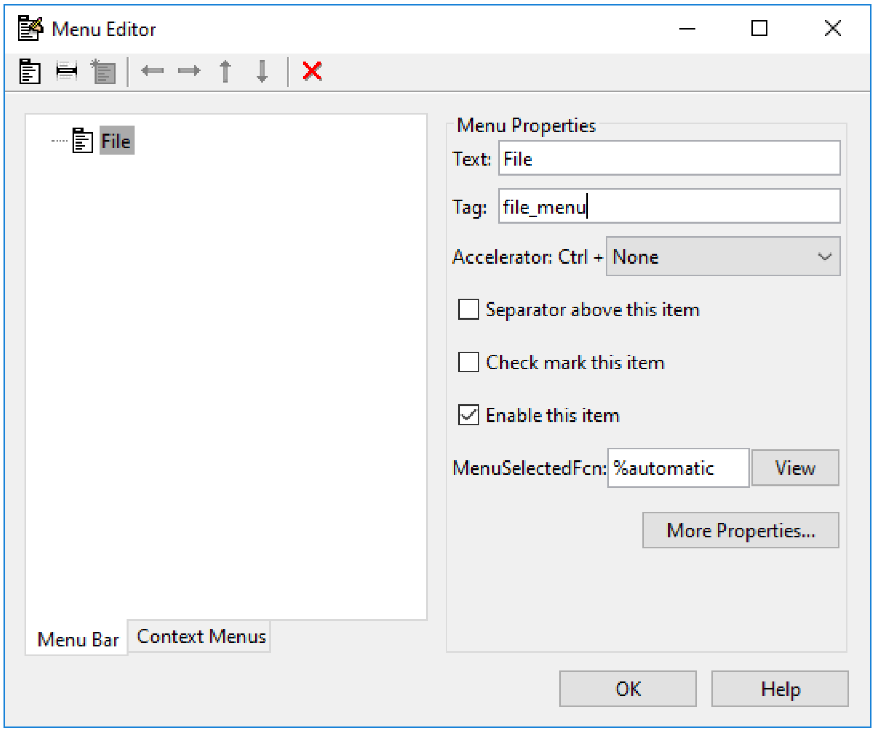
Create a Menu

1 Start a new menu by clicking the New Menu button in the toolbar. A menu title, `Untitled 1`, appears in the left pane of the dialog box.



By default, GUIDE selects the **Menu Bar** tab when you open the Menu Editor.

2 Click the menu title to display a selection of menu properties in the right pane.



Text: File

Tag: file_menu

Accelerator: Ctrl + None

□ Separator above this item

□ Check mark this item

☑ Enable this item

MenuSelectedFcn: %automatic

3 Fill in the **Text** and **Tag** fields for the menu. For example, set **Text** to `File` and set **Tag** to `file_menu`. Click outside the field for the change to take effect.

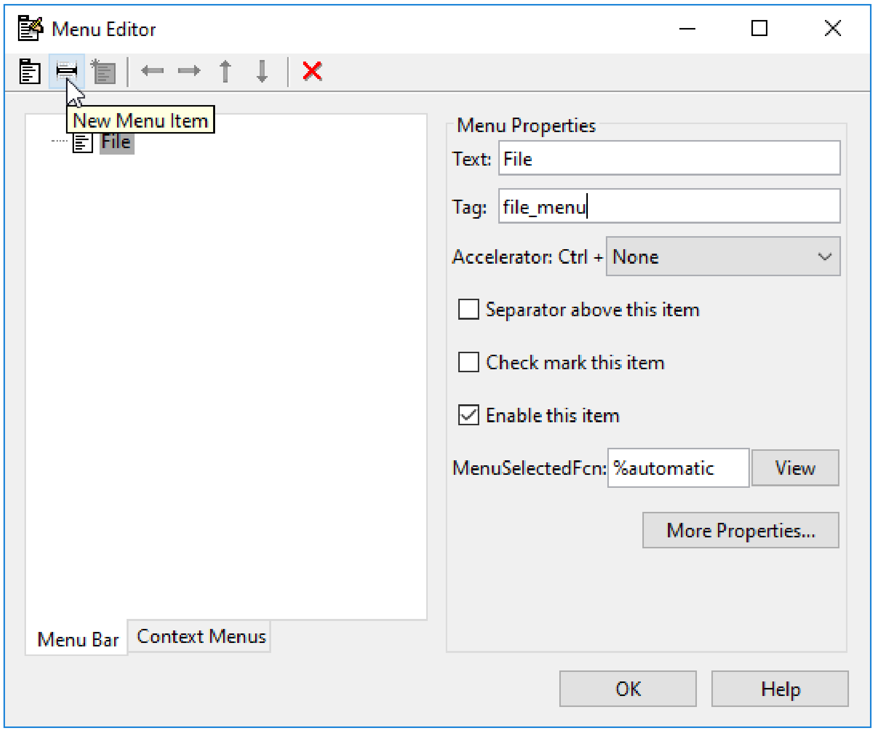
Text is a text label for the menu item. To display the & character in a label, use two & characters. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as labels, prepend a backslash character (\). For example, `\remove` yields **remove.**

Tag is a character vector that serves as an identifier for the menu object. It is used in the code to identify the menu item and must be unique in your code file.

Add Items to a Menu

Use the **New Menu Item** tool to create menu items that are displayed in the drop-down menu.

1 Add an **Open** menu item under `File`, by selecting `File` then clicking the **New Menu Item** button in the toolbar. A temporary numbered menu item label, `Untitled`, appears.



Menu Properties

Text: File

Tag: file_menu

Accelerator: Ctrl + None

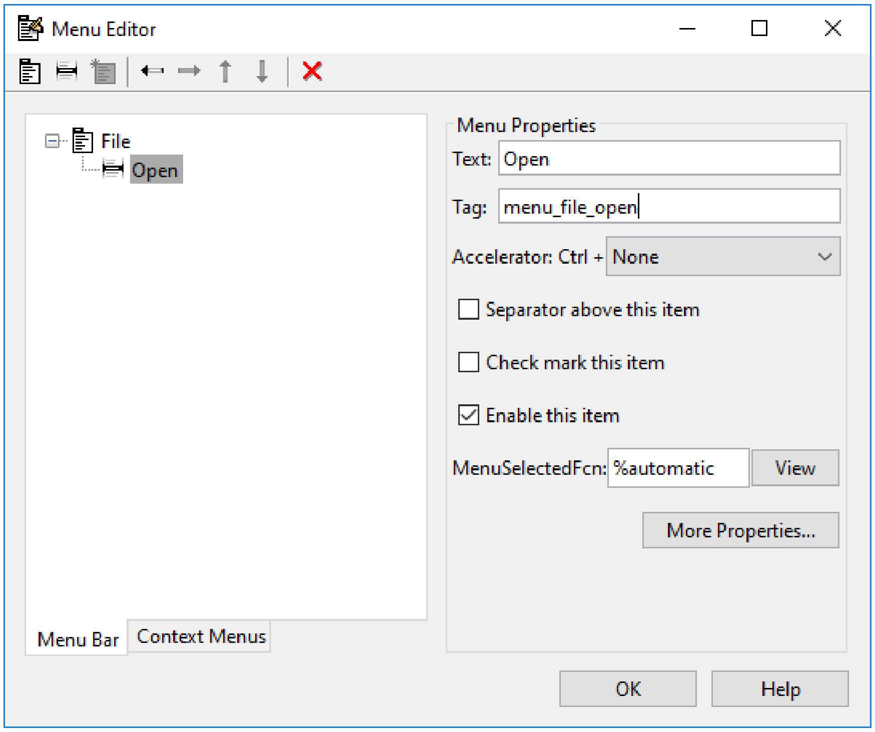
□ Separator above this item

□ Check mark this item

☑ Enable this item

MenuSelectedFcn: %automatic

2 Fill in the **Text** and **Tag** fields for the new menu item. For example, set **Text** to `Open` and set **Tag** to `menu_file_open`. Click outside the field for the change to take effect.



Menu Properties

Text: Open

Tag: menu_file_open

Accelerator: Ctrl + None

□ Separator above this item

□ Check mark this item

☑ Enable this item

MenuSelectedFcn: %automatic

You can also

- Choose an alphabetic keyboard accelerator for the menu item with the **Accelerator** pop-up menu. In combination with **Ctrl**, this is the keyboard equivalent for a menu item that does not have a child menu. Note that some accelerators may be used for other purposes on your system and that other actions may result.
- Display a separator above the menu item by checking **Separator above this item**.
- Display a check next to the menu item when the menu is first opened by checking **Check mark this item**. A check indicates the current state of the menu item. See the example in “Add Items to the Context Menu” on page 18-49.
- Enable this item when the menu is first opened by checking **Enable this item**. This allows the user to select this item when the menu is first opened. If you clear this option, the menu item appears dimmed when the menu is first opened, and the user cannot select it.
- Specify the **Callback** function that executes when the users selects the menu item. If you have not yet saved the UI, the default value is `%automatic`. When you save the UI, and if you have not changed this field, GUIDE automatically sets the value using a combination of the **Tag** field and the UI file name. See “Menu Item” on page 19-21 for more information about specifying this field and for programming menu items.

  The **View** button displays the callback, if there is one, in an editor. If you have not yet saved the UI, GUIDE prompts you to save it.
- Open the Property Inspector, where you can change all menu properties, by clicking the **More Properties** button. For detailed information about the properties, see Menu Properties.

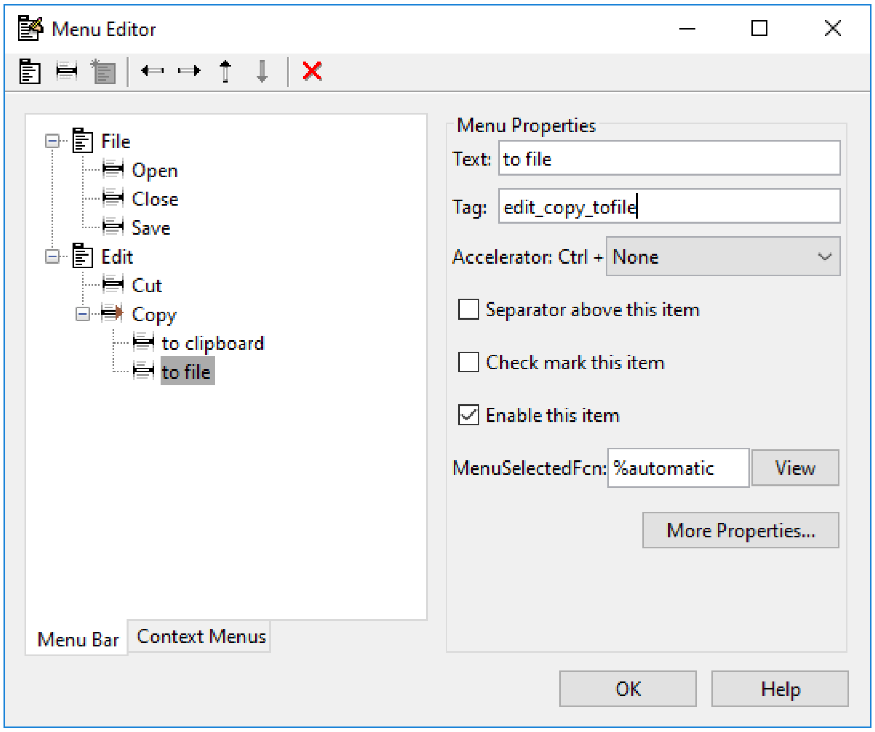
See “Menu Item” on page 19-21 and “How to Update a Menu Item Check” on page 19-23 for programming information and basic examples.

Additional Drop-Down Menus

To create additional drop-down menus, use the New Menu button in the same way you did to create the `File` menu. For example, the following figure also shows an `Edit` drop-down menu.

Cascading Menus

To create a cascading menu, select the menu item that will be the title for the cascading menu, then click the **New Menu Item** button. In the example below, `Edit` is a cascading menu.



Menu Properties

Text: to file

Tag: edit_copy_tofile

Accelerator: Ctrl + None

☐ Separator above this item

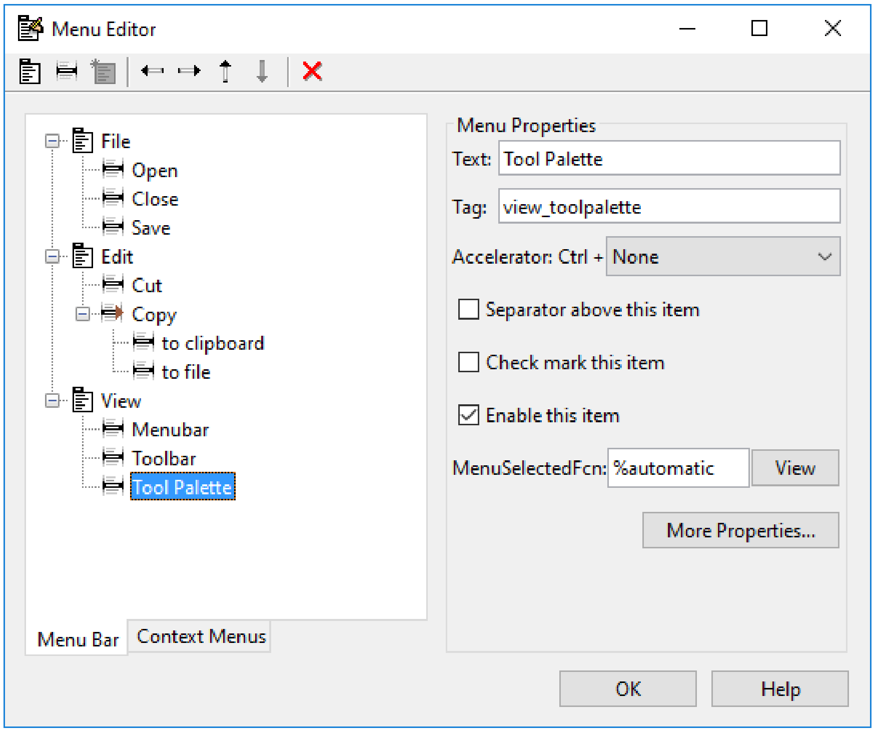
☐ Check mark this item

☑ Enable this item

MenuSelectedFcn: %automatic

See “Menu Item” on page 19-21 for information about programming menu items.

The following Menu Editor illustration shows three menus defined for the figure menu bar.



Menu Properties

Text: Tool Palette

Tag: view_toolpalette

Accelerator: Ctrl + None

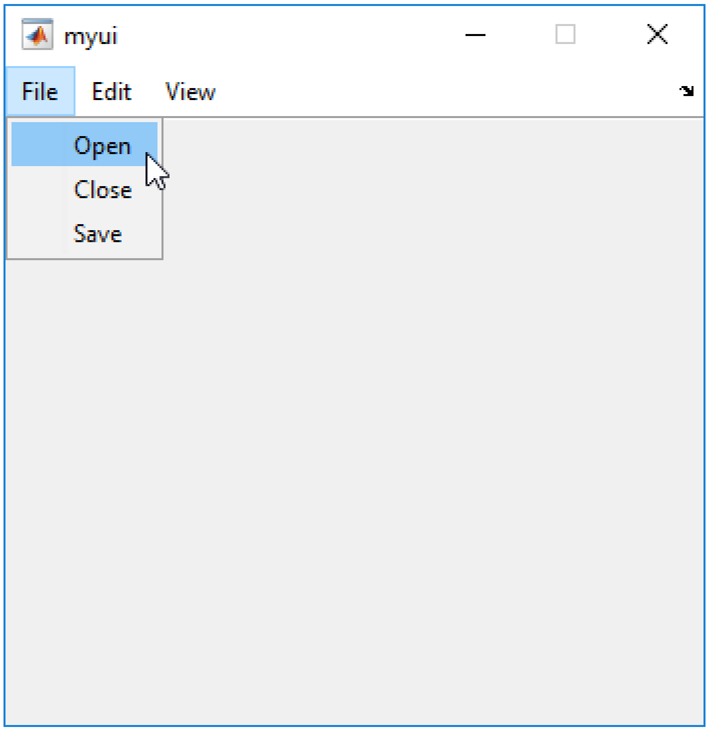
☐ Separator above this item

☐ Check mark this item

☑ Enable this item

MenuSelectedFcn: %automatic

When you run the app, the menu titles appear in the menu bar.



Context Menus

A context menu is displayed when a user right-clicks the object for which the menu is defined. The Menu Editor enables you to define context menus and associate them with objects in the layout. The process has three steps:

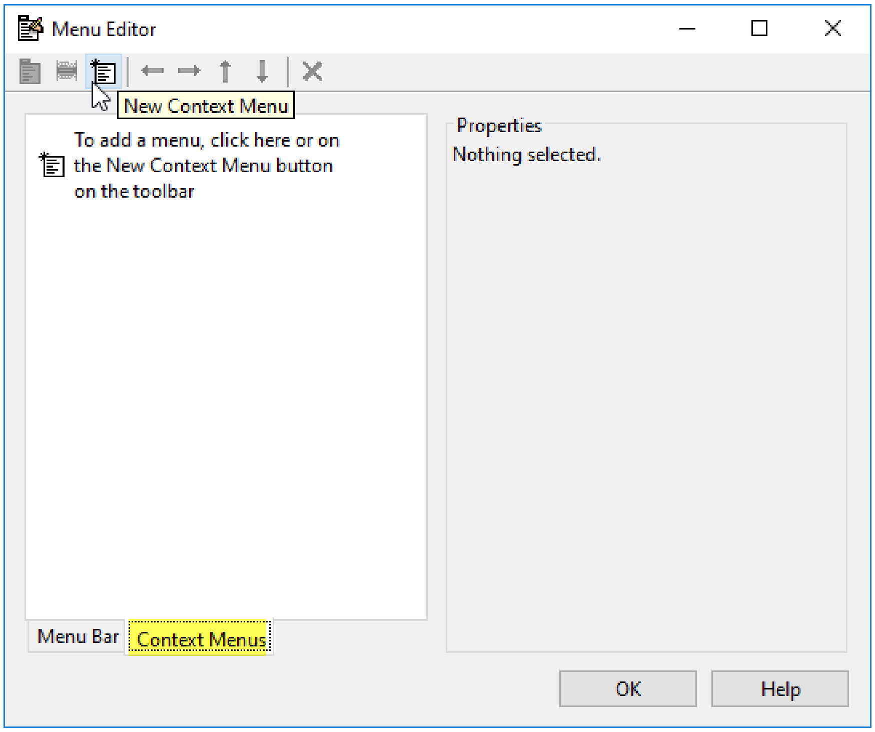
1. “Create the Parent Menu” on page 18-48
2. “Add Items to the Context Menu” on page 18-49
3. “Associate the Context Menu with an Object” on page 18-52

See “Menus for the Menu Bar” on page 18-41 for information about defining menus in general. See “Menu Item” on page 19-21 for information about defining local callback functions for your menus.

Create the Parent Menu

All items in a context menu are children of a menu that is not displayed on the figure menu bar. To define the parent menu:

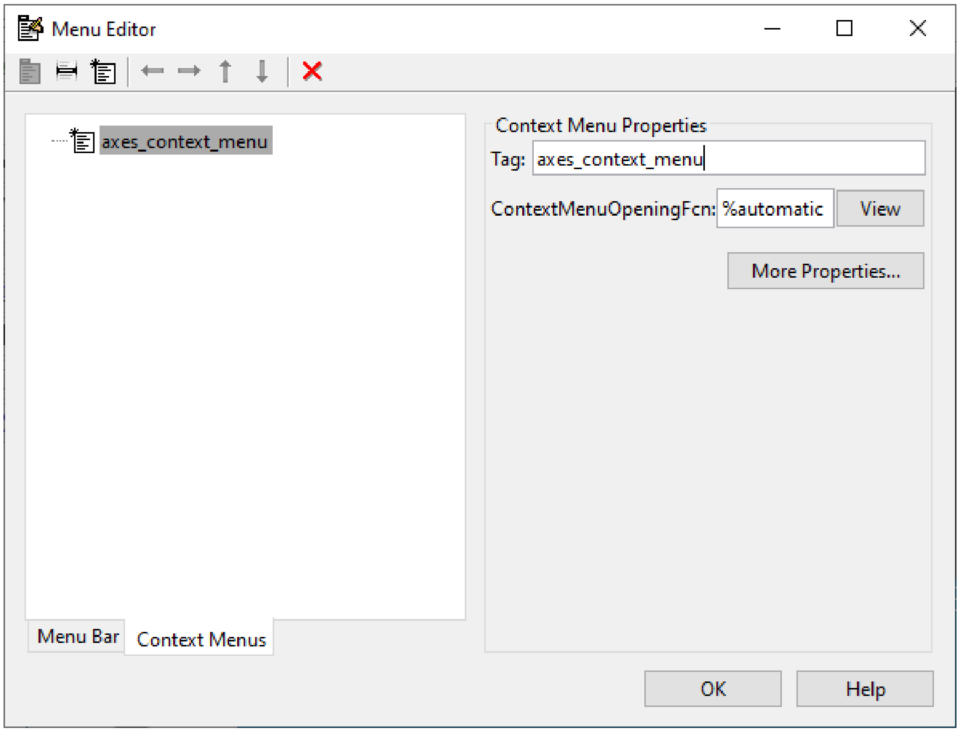
1. Select the Menu Editor's **Context Menus** tab and select the New Context Menu button from the toolbar.



To add a menu, click here or on the New Context Menu button on the toolbar

Nothing selected.

2 Select the menu, and in the **Tag** field type the context menu tag (`axes_context_menu` in this example).



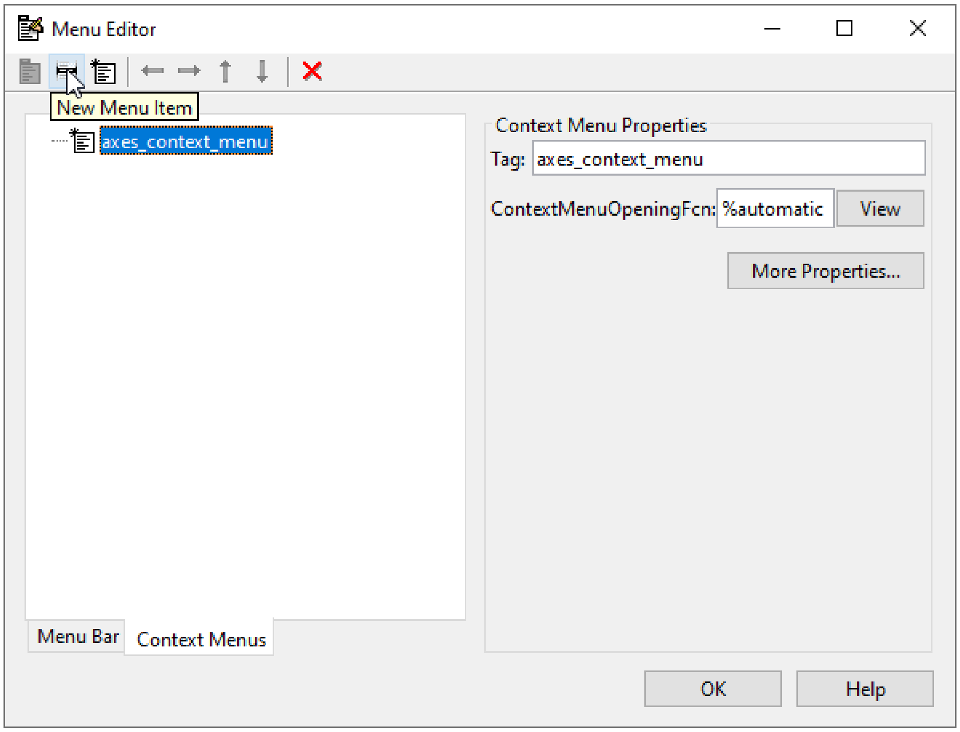
Tag: axes_context_menu

ContextMenuOpeningFcn: %automatic

Add Items to the Context Menu

Use the New Menu Item button to create menu items that are displayed in the context menu.

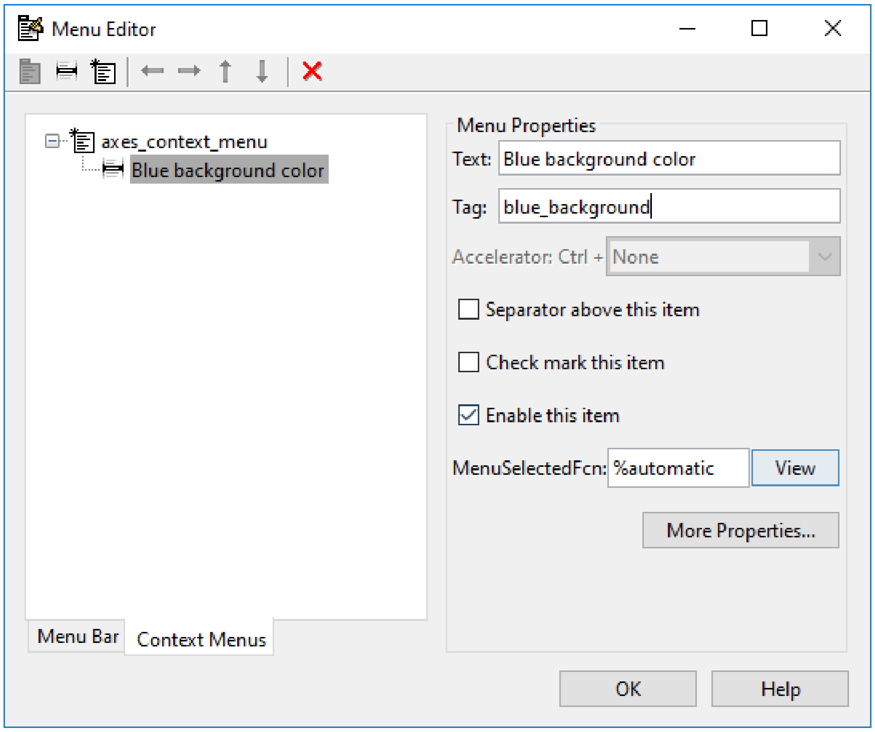
1 Add a `Blue background color` menu item to the menu by selecting `axes_context_menu` and clicking the **New Menu Item** tool. A temporary numbered menu item label, `Untitled`, appears.



Tag: axes_context_menu

ContextMenuOpeningFcn: %automatic

2 Fill in the **Text** and **Tag** fields for the new menu item. For example, set **Text** to `Blue background color` and set **Tag** to `blue_background`. Click outside the field for the change to take effect.



axes_context_menu
Blue background color

Menu Properties

Text: Blue background color

Tag: blue_background

Accelerator: Ctrl + None

☐ Separator above this item

☐ Check mark this item

☑ Enable this item

MenuSelectedFcn: %automatic

You can also modify menu items in these ways:

- Display a separator above the menu item by checking **Separator above this item**.
- Display a check next to the menu item when the menu is first opened by checking **Check mark this item**. A check indicates the current state of the menu item. See the example in “Add Items to the Context Menu” on page 18-49. See “How to Update a Menu Item Check” on page 19-23 for a code example.
- Enable this item when the menu is first opened by checking **Enable this item**. This allows the user to select this item when the menu is first opened. If you clear this option, the menu item appears dimmed when the menu is first opened, and the user cannot select it.
- Specify a **Callback** for the menu that performs the action associated with the menu item. If you have not yet saved the UI, the default value is `%automatic`. When you save the UI, and if you have not changed this field, GUIDE automatically creates a callback in the code file using a combination of the **Tag** field and the UI file name. The callback's name does not display in the **Callback** field of the Menu Editor, but selecting the menu item does trigger it.

  You can also type a command into the **Callback** field. It can be any valid MATLAB expression or command. For example, this command

  ```
  set(gca, 'Color', 'y')
  ```

  sets the current axes background color to yellow. However, the preferred approach to performing this operation is to place the callback in the code file. This avoids the use of `gca`, which is not always reliable when several figures or axes exist. Here is a version of this callback coded as a function in the code file:

  ```
  function axesyellow_Callback(hObject, eventdata, handles)
  % hObject    handle to axesyellow (see GCBO)
  % eventdata  reserved - to be defined in a future version of MATLAB
  ```

```
% handles    structure with handles and user data (see GUIDATA)
set(handles.axes1,'Color','y')
```

This code sets the background color of the axes with Tag `axes1` no matter to what object the context menu is attached to.

If you enter a callback value in the Menu Editor, it overrides the callback for the item in the code file, if any has been saved. If you delete a value that you entered in the **Callback** field, the callback for the item in the code file is executed when the user selects that item in the UI.

See “Menu Item” on page 19-21 for more information about specifying this field and for programming menu items.

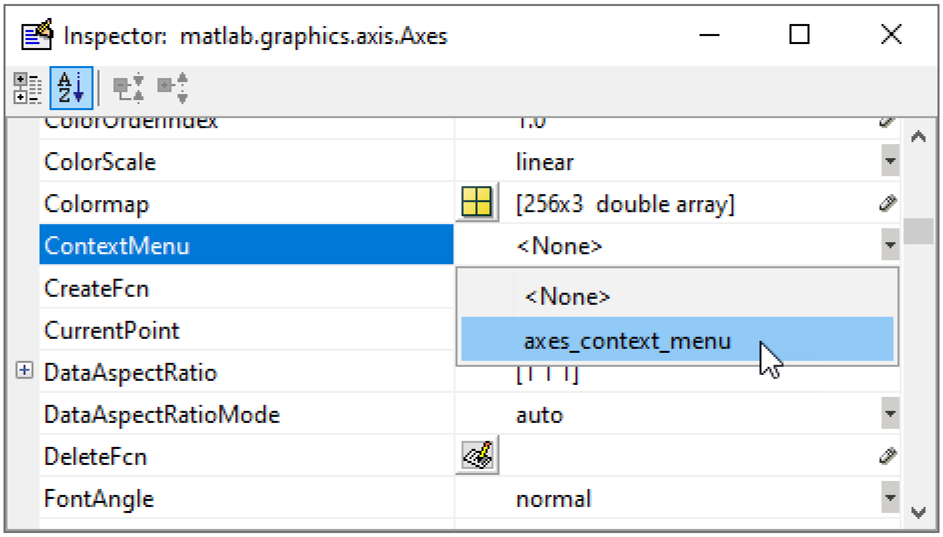
The **View** button displays the callback, if there is one, in an editor. If you have not yet saved the UI, GUIDE prompts you to save it.

- Open the Property Inspector, where you can change all menu properties except callbacks, by clicking the **More Properties** button. For detailed information about these properties, see ContextMenu Properties.

Associate the Context Menu with an Object

1. In the Layout Editor, select the object for which you are defining the context menu.
2. Use the Property Inspector to set this object's `ContextMenu` property to the name of the desired context menu.

The following figure shows the `ContextMenu` property for the `axes` object with `Tag` property `axes1`.



Inspector: matlab.graphics.axis.Axes

ColorOrderIndex	1.0
ColorScale	linear
Colormap	[256x3 double array]
ContextMenu	<None>
	<None> axes_context_menu
CreateFcn	
CurrentPoint	
DataAspectRatio	[1 1 1]
DataAspectRatioMode	auto
DeleteFcn	
FontAngle	normal

In the code file, complete the local callback function for each item in the context menu. Each callback executes when a user selects the associated context menu item. See “Menu Item” on page 19-21 for information on defining the syntax.

See “How to Update a Menu Item Check” on page 19-23 for programming information and basic examples.

See Also

Related Examples

- “Write Callbacks in GUIDE” on page 19-2
- “Callbacks for Specific Components” on page 19-14
- “App Building Components” on page 4-2

19

Programming a GUIDE App

Write Callbacks in GUIDE

In this section...

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 3-7 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

Callbacks for Different User Actions

UI and graphics components have certain properties that you can associate with specific callback functions. Each of these properties corresponds to a specific user action. For example, a uicontrol has a property called `Callback`. You can set the value of this property to be a handle to a callback function, an anonymous function, or a character vector containing a MATLAB expression. Setting this property makes your app respond when the user interacts with the uicontrol. If the `Callback` property has no specified value, then nothing happens when the user interacts with the uicontrol.

This table lists the callback properties that are available, the user actions that trigger the callback function, and the most common UI and graphics components that use them.

Callback Property	User Action	Components That Use This Property
`ButtonDownFcn`	End user presses a mouse button while the pointer is on the component or figure.	`axes`, `figure`, `uibuttongroup`, `uicontrol`, `uipanel`, `uitable`,
`Callback`	End user triggers the component. For example: selecting a menu item, moving a slider, or pressing a push button.	`uicontextmenu`, `uicontrol`, `uimenu`
`CellEditCallback`	End user edits a value in a table whose cells are editable.	`uitable`
`CellSelectionCallback`	End user selects cells in a table.	`uitable`

Callback Property	User Action	Components That Use This Property
ClickedCallback	End user clicks the push tool or toggle tool with the left mouse button.	uitoggletool, uipushtool
CloseRequestFcn	The figure closes.	figure
CreateFcn	Callback executes when MATLAB creates the object, but before it is displayed.	axes, figure, uibuttongroup, uicontextmenu, uicontrol, uimenu, uipushtool, uipanel, uitable, uitoggletool, uitoolbar
DeleteFcn	Callback executes just before MATLAB deletes the figure.	axes, figure, uibuttongroup, uicontextmenu, uicontrol, uimenu, uipushtool, uipanel, uitable, uitoggletool, uitoolbar
KeyPressFcn	End user presses a keyboard key while the pointer is on the object.	figure, uicontrol, uipanel, uipushtool, uitable, uitoolbar
KeyReleaseFcn	End user releases a keyboard key while the pointer is on the object.	figure, uicontrol, uitable
OffCallback	Executes when the State of a toggle tool changes to 'off'.	uitoggletool
OnCallback	Executes when the State of a toggle tool changes to 'on'.	uitoggletool
SizeChangedFcn	End user resizes a button group, figure, or panel whose Resize property is 'on'.	figure, uipanel, uibuttongroup
SelectionChangedFcn	End user selects a different radio button or toggle button within a button group.	uibuttongroup
WindowButtonDownFcn	End user presses a mouse button while the pointer is in the figure window.	figure
WindowButtonMotionFcn	End user moves the pointer within the figure window.	figure
WindowButtonUpFcn	End user releases a mouse button.	figure
WindowKeyPressFcn	End user presses a key while the pointer is on the figure or any of its child objects.	figure
WindowKeyReleaseFcn	End user releases a key while the pointer is on the figure or any of its child objects.	figure
WindowScrollWheelFcn	End user turns the mouse wheel while the pointer is on the figure.	figure

GUIDE-Generated Callback Functions and Property Values

How GUIDE Manages Callback Functions and Properties

After you add a `uicontrol`, `uimenu`, or `uicontextmenu` component to your UI, but before you save it, GUIDE populates the `Callback` property with the value, `%automatic`. This value indicates that GUIDE will generate a name for the callback function.

When you save your UI, GUIDE adds an empty callback function definition to your code file, and it sets the control's `Callback` property to be an anonymous function. This function definition is an example of a GUIDE-generated callback function for a push button.

```
function pushbutton1_Callback(hObject,eventdata,handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

end
```

If you save this UI with the name, `myui`, then GUIDE sets the push button's `Callback` property to the following value:

```
@(hObject,eventdata)myui('pushbutton1_Callback',hObject,eventdata,guidata(hObject))
```

This is an anonymous function that serves as a reference to the function, `pushbutton1_Callback`. This anonymous function has four input arguments. The first argument is the name of the callback function. The last three arguments are provided by MATLAB, and are discussed in the section, "GUIDE Callback Syntax" on page 19-4.

Note GUIDE does not automatically generate callback functions for other UI components, such as tables, panels, or button groups. If you want any of these components to execute a callback function, then you must create the callback by right-clicking on the component in the layout, and selecting an item under **View Callbacks** in the context menu.

GUIDE Callback Syntax

All callbacks must accept at least three input arguments:

- `hObject` — The UI component that triggered the callback.
- `eventdata` — A variable that contains detailed information about specific mouse or keyboard actions.
- `handles` — A `struct` that contains all the objects in the UI. GUIDE uses the `guidata` function to store and maintain this structure.

For the callback function to accept additional arguments, you must put the additional arguments at the end of the argument list in the function definition.

The eventdata Argument

The `eventdata` argument provides detailed information to certain callback functions. For example, if the end user triggers the `KeyPressFcn`, then MATLAB provides information regarding the specific key (or combination of keys) that the end user pressed. If `eventdata` is not available to the callback function, then MATLAB passes it as an empty array. The following table lists the callbacks and components that use `eventdata`.

Callback Property Name	Component

`WindowKeyPressFcn`
`WindowKeyReleaseFcn`
`WindowScrollWheel`
	`figure`

`KeyPressFcn`	`figure`, `uicontrol`, `uitable`
`KeyReleaseFcn`	`figure`, `uicontrol`, `uitable`
`SelectionChangedFcn`	`uibuttongroup`
`CellEditCallback` `CellSelectionCallback`	`uitable`

Share Data Among GUIDE Callbacks

To create controls, menus, and graphics objects in your app that are interdependent, you must explicitly share data with the parts of your app that need to access the component.

Method	Description	Requirements and Trade-Offs
Share UserData	Get or set property values directly through the component object. All UI components have a `UserData` property that can store any MATLAB data.	• Requires access to the component to set or retrieve the properties. • `UserData` holds only one variable at a time, but you can store multiple values as a `struct` array or cell array.
Share Application Data	Associate data with a specific component using the `setappdata` function. You can access it later using the `getappdata` function.	• Requires access to the component to set or retrieve the application data. • Can share multiple variables.
Use `guidata`	Share data with the figure window using the `guidata` function.	• Stores or retrieves the data through any UI component. • Stores only one variable at a time, but you can store multiple values as a `struct` array or cell array.

Share UserData in GUIDE Apps

UI components contain useful information in their properties. For example, you can find the current position of a slider by querying its `Value` property. In addition, all components have a `UserData` property, which can store any MATLAB variable. All callback functions can access the value stored in the `UserData` property as long as those functions can access the component.

To set up a GUIDE app for sharing slider data with the `UserData` property, perform these steps:

1. In the Command Window, type `guide` to open a new blank GUI.
2. Display the names of the UI components in the component palette:

   a. Select **File** > **Preferences** > **GUIDE**.

   b. Select **Show names in component palette**.

   c. Click **OK**.

3. Select the push button tool from the component palette at the left side of the Layout Editor and drag it into the layout area.
4. Select the slider tool from the component palette at the left side of the Layout Editor and drag it into the layout area.
5. Select **File** > **Save**. Save the UI as `myslider.fig`. MATLAB opens the code file in the Editor.
6. Set the initial value of the `UserData` property in the opening function, `myslider_OpeningFcn`. This function executes just before the UI is visible to users.

   In `myslider_OpeningFcn`, insert these commands immediately after the command, `handles.output = hObject`.

   ```
   data = struct('val',0,'diffMax',1);
   set(handles.slider1,'UserData',data);
   ```

   After you add the commands, `myslider_OpeningFcn` looks like this.

   ```
   function myslider_OpeningFcn(hObject, eventdata, handles, varargin)
   % This function has no output args, see OutputFcn.
   % hObject    handle to figure
   % eventdata  reserved - to be defined in a future version of MATLAB
   % handles    structure with handles and user data (see GUIDATA)
   % varargin   command line arguments to junk (see VARARGIN)

   % Choose default command line output for myslider
   handles.output = hObject;
   data = struct('val',0,'diffMax',1);
   set(handles.slider1,'UserData',data);

   % Update handles structure
   guidata(hObject, handles);

   % UIWAIT makes myslider wait for user response
   % uiwait(handles.figure1);
   ```

   Notice that `handles` is an input argument to `myslider_OpeningFcn`. The `handles` variable is a structure that contains all the components in the UI. Each field in this structure corresponds to a separate component. Each field name matches the `Tag` property of the corresponding component. Thus, `handles.slider1` is the slider component in this UI. The command, `set(handles.slider1,'UserData',data)` stores the variable, `data`, in the `UserData` property of the slider.
7. Add code to the slider callback for modifying the data. Add these commands to the end of the function, `slider1_Callback`.

   ```
   maxval = get(hObject,'Max');
   sval = get(hObject,'Value');
   diffMax = maxval - sval;
   data = get(hObject,'UserData');
   data.val = sval;
   data.diffMax = diffMax;
   % Store data in UserData of slider
   set(hObject,'UserData',data);
   ```

   After you add the commands, `slider1_Callback` looks like this.

   ```
   % --- Executes on slider movement.
   function slider1_Callback(hObject, eventdata, handles)
   ```

```
% hObject    handle to slider1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
maxval = get(hObject,'Max');
sval = get(hObject,'Value');
diffMax = maxval - sval;
data = get(hObject,'UserData');
data.val = sval;
data.diffMax = diffMax;
% Store data in UserData of slider
set(hObject,'UserData',data);
```

Notice that `hObject` is an input argument to the `slider1_Callback` function. `hObject` is always the component that triggers the callback (the slider, in this case). Thus, `set(hObject,'UserData',data)`, stores the `data` variable in the `UserData` property of the slider.

8. Add code to the push button callback for retrieving the data. Add these commands to the end of the function, `pushbutton1_Callback`.

   ```
   % Get UserData from the slider
   data = get(handles.slider1,'UserData');
   currentval = data.val;
   diffval = data.diffMax;
   display([currentval diffval]);
   ```

   After you add the commands, `pushbutton1_Callback` looks like this.

   ```
   % --- Executes on button press in pushbutton1.
   function pushbutton1_Callback(hObject, eventdata, handles)
   % hObject    handle to pushbutton1 (see GCBO)
   % eventdata  reserved - to be defined in a future version of MATLAB
   % handles    structure with handles and user data (see GUIDATA)

   % Get UserData from the slider
   data = get(handles.slider1,'UserData');
   currentval = data.val;
   diffval = data.diffMax;
   display([currentval diffval]);
   ```

   This code uses the handles structure to access the slider. The command, `data = get(handles.slider1,'UserData')`, gets the slider's `UserData` property. Then, the `display` function displays the stored values.

9. Save your code by pressing **Save** in the Editor Toolstrip.

Share Application Data in GUIDE Apps

To store application data, call the `setappdata` function:

```
setappdata(obj,name,value);
```

The first input, `obj`, is the component object in which to store the data. The second input, `name`, is a friendly name that describes the value. The third input, `value`, is the value you want to store.

To retrieve application data, use the `getappdata` function:

```
data = getappdata(obj,name);
```

The component, `obj`, must be the component object containing the data. The second input, `name`, must match the name you used to store the data. Unlike the `UserData` property, which only holds only one variable, you can use `setappdata` to store multiple variables.

To set up a GUIDE app for sharing application data, perform these steps:

1. In the Command Window, type `guide` to open a new blank GUI.
2. Display the names of the UI components in the component palette:

   a. Select **File** > **Preferences** > **GUIDE**.

   b. Select **Show names in component palette**.

   c. Click **OK**.
3. Select the push button tool from the component palette at the left side of the Layout Editor and drag it into the layout area.
4. Select the slider tool from the component palette at the left side of the Layout Editor and drag it into the layout area.
5. Select **File** > **Save**. Save the UI as `myslider.fig`. MATLAB opens the code file in the Editor.
6. Set the initial value of the application data in the opening function, `myslider_OpeningFcn`. This function executes just before the UI is visible to users. In `myslider_OpeningFcn`, insert these commands immediately after the command, `handles.output = hObject`.

   ```
   setappdata(handles.figure1,'slidervalue',0);
   setappdata(handles.figure1,'difference',1);
   ```

   After you add the commands, `myslider_OpeningFcn` looks like this.

   ```
   function myslider_OpeningFcn(hObject,eventdata,handles,varargin)
   % This function has no output args, see OutputFcn.
   % hObject    handle to figure
   % eventdata  reserved - to be defined in a future version of MATLAB
   % handles    structure with handles and user data (see GUIDATA)
   % varargin   command line arguments to junk (see VARARGIN)

   % Choose default command line output for junk
   handles.output = hObject;
   setappdata(handles.figure1,'slidervalue',0);
   setappdata(handles.figure1,'difference',1);

   % Update handles structure
   guidata(hObject, handles);

   % UIWAIT makes junk wait for user response (see UIRESUME)
   % uiwait(handles.figure1);
   ```

   Notice that `handles` is an input argument to `myslider_OpeningFcn`. The `handles` variable is a structure that contains all the components in the UI. Each field in this structure corresponds to a separate component. Each field name matches the `Tag` property of the corresponding component. In this case, `handles.figure1` is the figure object. Thus, `setappdata` can use this figure object to store the data.
7. Add code to the slider callback for changing the data. Add these commands to the end of the function, `slider1_Callback`.

```
maxval = get(hObject,'Max');
currval = get(hObject,'Value');
diffMax = maxval - currval;
% Store application data
setappdata(handles.figure1,'slidervalue',currval);
setappdata(handles.figure1,'difference',diffMax);
```

After you add the commands, `slider1_Callback` looks like this.

```
function slider1_Callback(hObject, eventdata, handles)
% hObject    handle to slider1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
maxval = get(hObject,'Max');
currval = get(hObject,'Value');
diffMax = maxval - currval;
% Store application data
setappdata(handles.figure1,'slidervalue',currval);
setappdata(handles.figure1,'difference',diffMax);
```

This callback function has access to the `handles` structure, so the `setappdata` commands store the data in `handles.figure1`.

8 Add code to the push button callback for retrieving the data. Add these commands to the end of the function, `pushbutton1_Callback`.

```
% Retrieve application data
currentval = getappdata(handles.figure1,'slidervalue');
diffval = getappdata(handles.figure1,'difference');
display([currentval diffval]);
```

After you add the commands, `pushbutton1_Callback` looks like this.

```
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Retrieve application data
currentval = getappdata(handles.figure1,'slidervalue');
diffval = getappdata(handles.figure1,'difference');
display([currentval diffval]);
```

This callback function has access to the `handles` structure, so the `getappdata` commands retrieve the data from `handles.figure1`.

9 Save your code by pressing **Save** in the Editor Toolstrip.

Use guidata to Store and Share Data in GUIDE Apps

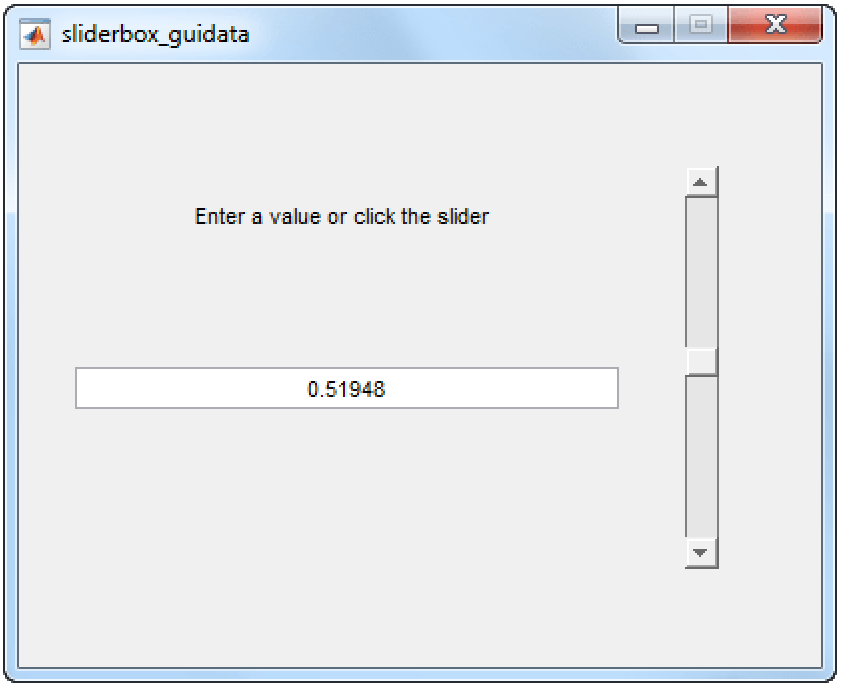
GUIDE uses the `guidata` function to store a structure called `handles`, which contains all the UI components. MATLAB passes the `handles` array to every callback function. If you want to use `guidata` to share additional data, then add fields to the `handles` structure in the opening function. The opening function is a function defined near the top of your code file that has `_OpeningFcn` in the name.

To modify your data in a callback function, modify the handles structure, and then store it using the guidata function. This slider callback function shows how to modify and store the handles structure in a GUIDE callback function.

```
function slider1_Callback(hObject, eventdata,handles)
% hObject    handle to slider1 (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range
    handles.myvalue = 2;
    guidata(hObject,handles);
end
```

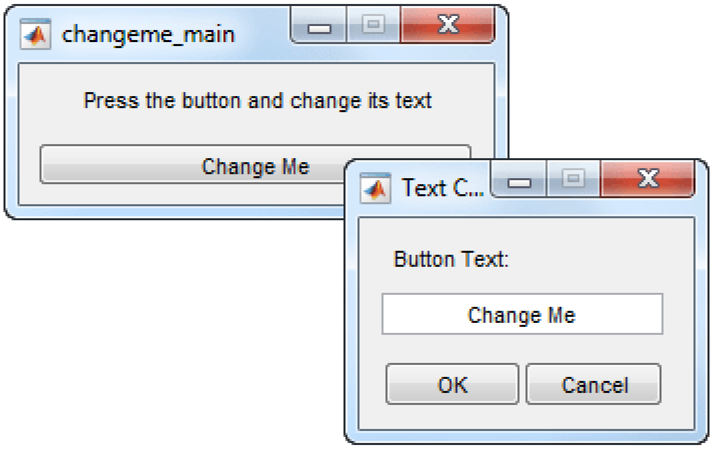
GUIDE Example: Share Slider Data Using guidata

Here is a prebuilt GUIDE app that uses the guidata function to share data between a slider and a text field. When you move the slider, the number displayed in the text field changes to show the new slider position.



GUIDE Example: Share Data Between Two Apps

Here is a prebuilt GUIDE app that uses application data and the guidata function to share data between two dialog boxes. When you enter text in the second dialog box and click **OK**, the button label changes in the first dialog box.



In changeme_main.m, the buttonChangeMe_Callback function executes this command to display the second dialog box:

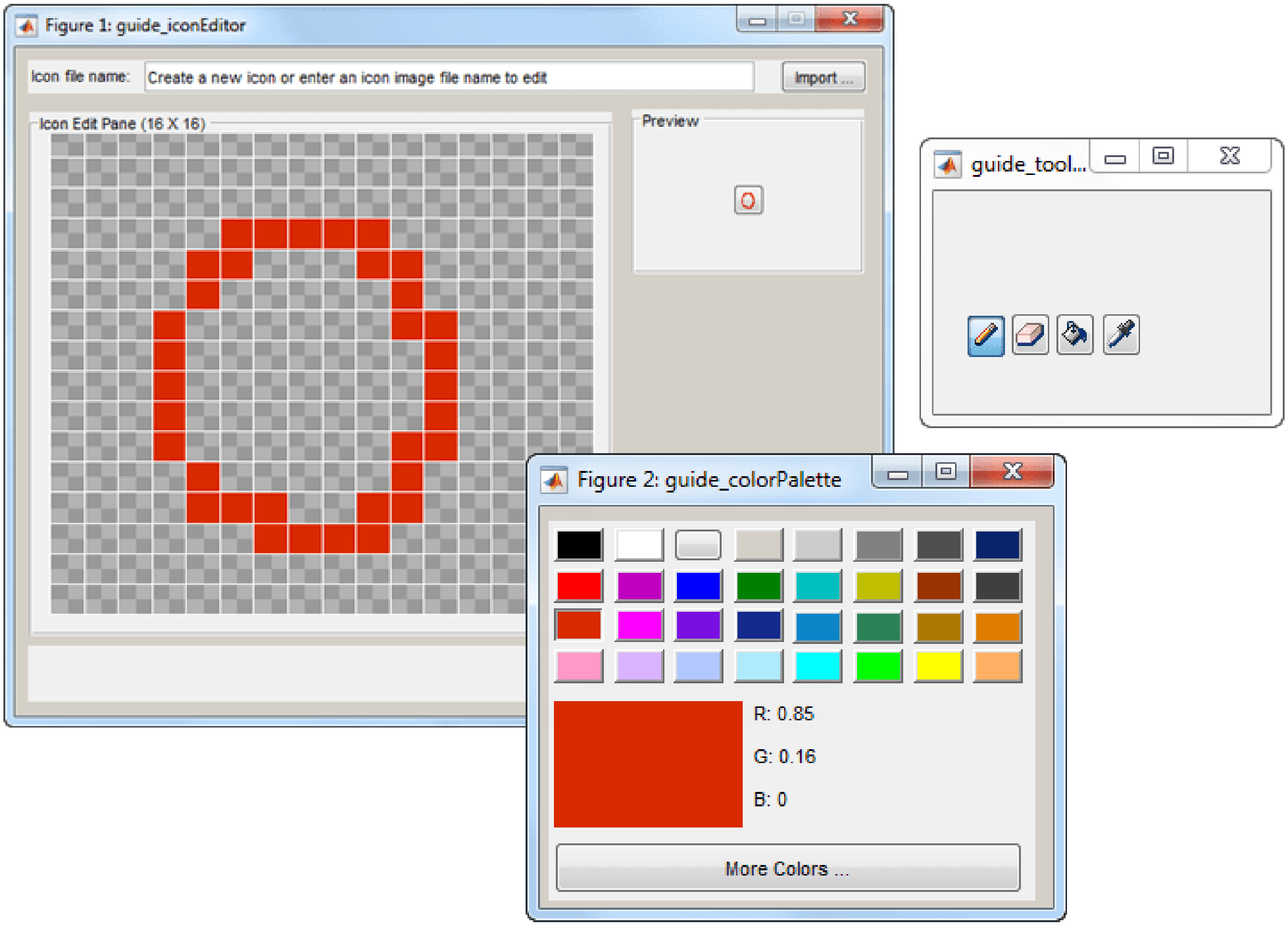
```
changeme_dialog('changeme_main', handles.figure)
```

The handles.figure input argument is the Figure object for the **changeme_main** dialog box.

The changeme_dialog function retrieves the handles structure from the Figure object. Thus, the entire set of components in the **changeme_main** dialog box is available to the second dialog box.

GUIDE Example: Share Data Among Three Apps

Here is a prebuilt GUIDE app that uses guidata and UserData to share data among three app windows. The large window is an icon editor that accepts information from the tool palette and color palette windows.



Icon file name: Create a new icon or enter an icon image file name to edit

R: 0.85

G: 0.16

B: 0

In `guide_inconeditor.m`, the function `guide_iconeditor_OpeningFcn` contains this command:

```
colorPalette = guide_colorpalette('iconEditor', hObject)
```

The arguments are:

- `'iconEditor'` specifies that a callback in the **guide_iconEditor** window triggered the execution of the function.
- `hObject` is the `Figure` object for the **guide_iconEditor** window.
- `colorPalette` is the `Figure` object for the **guide_colorPalette** window.

Similarly, `guide_iconeditor_OpeningFcn` calls the `guide_toolpalette` function with similar input and output arguments.

Passing the `Figure` object between these functions allows the **guide_iconEditor** window to access the `handles` structure of the other two windows. Likewise, the other two windows can access the `handles` structure for the **guide_iconEditor** window.

Renaming and Removing GUIDE-Generated Callbacks

Renaming Callbacks

GUIDE creates the name of a callback function by combining the component’s `Tag` property and the callback property name. If you change the component’s `Tag` value, then GUIDE changes the callback's name the next time you save the UI.

If you decide to change the `Tag` value after saving the UI, then GUIDE updates the following items (assuming that all components have unique `Tag` values).

- Component's callback function definition
- Component’s callback property value
- References in the code file to the corresponding field in the `handles` structure

To rename a callback function without changing the component’s `Tag` property:

1. Change the name in the callback function definition.
2. Update the component’s callback property by changing the first argument passed to the anonymous function. For example, the original callback property for a push button might look like this:

   ```
   @(hObject,eventdata)myui('pushbutton1_Callback',...
                               hObject,eventdata,guidata(hObject))
   ```

   In this example, you must change, `'pushbutton1_Callback'` to the new function name.
3. Change all other references to the old function name to the new function name in the code file.

Deleting Callbacks

You can delete a callback function when you want to remove or change the function that executes when the end user performs a specific action. To delete a callback function:

1. Search and replace all instances that refer to the callback function in your code.
2. Open the UI in GUIDE and replace all instances that refer to the callback function in the Property Inspector.
3. Delete the callback function.

See Also

Related Examples

- “Callbacks for Specific Components” on page 19-14
- “Anonymous Functions”
- “Share Data Among Callbacks” on page 11-9
- “Callbacks in App Designer” on page 6-16

Callbacks for Specific Components

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 3-7 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

Coding the behavior of a UI component involves specific tasks that are unique to the type of component you are working with. This topic contains simple examples of callbacks for each type of component. For general information about coding callbacks, see “Write Callbacks in GUIDE” on page 19-2 or “Create Callbacks for Apps Created Programmatically” on page 11-2.

How to Use the Example Code

If you are working in GUIDE, then right-click on the component in your layout and select the appropriate callback property from the **View Callbacks** menu. Doing so creates an empty callback function that is automatically associated with the component. The specific function name that GUIDE creates is based on the component’s `Tag` property, so your function name might be slightly different than the function name in the example code. Do not change the function name that GUIDE creates in your code. To use the example code in your app, copy the code from the example’s function body into your function’s body.

Push Button

This code is an example of a push button callback function in GUIDE. Associate this function with the push button `Callback` property to make it execute when the end user clicks on the push button.

```
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
display('Goodbye');
close(gcf);
```

The first line of code, `display('Goodbye')`, displays `'Goodbye'` in the Command Window. The next line gets the UI window using `gcf` and then closes it.

Toggle Button

This code is an example of a toggle button callback function in GUIDE. Associate this function with the toggle button `Callback` property to make it execute when the end user clicks on the toggle button.

```
function togglebutton1_Callback(hObject,eventdata,handles)
% hObject    handle to togglebutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of togglebutton1
button_state = get(hObject,'Value');
if button_state == get(hObject,'Max')
    display('down');
elseif button_state == get(hObject,'Min')
    display('up');
end
```

The toggle button's `Value` property matches the `Min` property when the toggle button is up. The `Value` changes to the `Max` value when the toggle button is depressed. This callback function gets the toggle button's `Value` property and then compares it with the `Max` and `Min` properties. If the button is depressed, then the function displays `'down'` in the Command Window. If the button is up, then the function displays `'up'`.

Radio Button

This code is an example of a radio button callback function in GUIDE. Associate this function with the radio button `Callback` property to make it execute when the end user clicks on the radio button.

```
function radiobutton1_Callback(hObject, eventdata, handles)
% hObject    handle to radiobutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of radiobutton1

if (get(hObject,'Value') == get(hObject,'Max'))
    display('Selected');
else
    display('Not selected');
end
```

The radio button's `Value` property matches the `Min` property when the radio button is not selected. The `Value` changes to the `Max` value when the radio button is selected. This callback function gets the radio button's `Value` property and then compares it with the `Max` and `Min` properties. If the button is selected, then the function displays `'Selected'` in the Command Window. If the button is not selected, then the function displays `'Not selected'`.

Note Use a button group to manage exclusive selection behavior for radio buttons. See "Button Group" on page 19-20 for more information.

Check Box

This code is an example of a check box callback function in GUIDE. Associate this function with the check box `Callback` property to make it execute when the end user clicks on the check box.

```
function checkbox1_Callback(hObject, eventdata, handles)
% hObject    handle to checkbox1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of checkbox1

if (get(hObject,'Value') == get(hObject,'Max'))
    display('Selected');
else
    display('Not selected');
end
```

The check box's `Value` property matches the `Min` property when the check box is not selected. The `Value` changes to the `Max` value when the check box is selected. This callback function gets the check box's `Value` property and then compares it with the `Max` and `Min` properties. If the check box is selected, the function displays `'Selected'` in the Command Window. If the check box is not selected, it displays `'Not selected'`.

Edit Text Field

This code is an example of a callback for an edit text field in GUIDE. Associate this function with the uicontrol's `Callback` property to make it execute when the end user types inside the text field.

```
function edit1_Callback(hObject, eventdata, handles)
% hObject    handle to edit1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit1 as text
%        str2double(get(hObject,'String')) returns contents as double
input = get(hObject,'String');
display(input);
```

When the user types characters inside the text field and presses the **Enter** key, the callback function retrieves those characters and displays them in the Command Window.

To enable users to enter multiple lines of text, set the `Max` and `Min` properties to numeric values that satisfy `Max - Min > 1`. For example, set `Max` to `2`, and `Min` to `0` to satisfy the inequality. In this case, the callback function triggers when the end user clicks on an area in the UI that is outside of the text field.

Retrieve Numeric Values

If you want to interpret the contents of an edit text field as numeric values, then convert the characters to numbers using the `str2double` function. The `str2double` function returns `NaN` for nonnumeric input.

This code is an example of an edit text field callback function that interprets the user's input as numeric values.

```
function edit1_Callback(hObject, eventdata, handles)
% hObject    handle to edit1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit1 as text
% str2double(get(hObject,'String')) returns contents as a double
input = str2double(get(hObject,'String'));
if isnan(input)
```

```
  errordlg('You must enter a numeric value','Invalid Input','modal')
  uicontrol(hObject)
  return
else
  display(input);
end
```

When the end user enters values into the edit text field and presses the **Enter** key, the callback function gets the value of the `String` property and converts it to a numeric value. Then, it checks to see if the value is `NaN` (nonnumeric). If the input is `NaN`, then the callback presents an error dialog box.

Slider

This code is an example of a slider callback function in GUIDE. Associate this function with the slider `Callback` property to make it execute when the end user moves the slider.

```
function slider1_Callback(hObject, eventdata, handles)
% hObject    handle to slider1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine...
slider_value = get(hObject,'Value');
display(slider_value);
```

When the end user moves the slider, the callback function gets the current value of the slider and displays it in the Command Window. By default, the slider’s range is [0, 1]. To modify the range, set the slider’s `Max` and `Min` properties to the maximum and minimum values, respectively.

List Box

Populate Items in the List Box

If you are developing an app using GUIDE, use the list box `CreateFcn` callback to add items to the list box.

This code is an example of a list box `CreateFcn` callback that populates the list box with the items, `Red`, `Green`, and `Blue`.

```
function listbox1_CreateFcn(hObject, eventdata, handles)
% hObject    handle to listbox1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns

% Hint: listbox controls usually have a white background on Windows.
if ispc && isequal(get(hObject,'BackgroundColor'), ...
    get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
set(hObject,'String',{'Red';'Green';'Blue'});
```

The last line, `set(hObject,'String',{'Red';'Green';'Blue'})`, populates the contents of the list box.

Change the Selected Item

When the end user selects a list box item, the list box’s Value property changes to a number that corresponds to the item’s position in the list. For example, a value of 1 corresponds to the first item in the list. If you want to change the selection in your code, then change the Value property to another number between 1 and the number of items in the list.

For example, you can use the handles structure in GUIDE to access the list box and change the Value property:

```
set(handles.listbox1,'Value',2)
```

The first argument, handles.listbox1, might be different in your code, depending on the value of the list box Tag property.

Write the Callback Function

This code is an example of a list box callback function in GUIDE. Associate this function with the list box Callback property to make it execute when a selects an item in the list box.

```
function listbox1_Callback(hObject, eventdata, handles)
% hObject    handle to listbox1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hints: contents = cellstr(get(hObject,'String')) returns contents
% contents{get(hObject,'Value')} returns selected item from listbox1
items = get(hObject,'String');
index_selected = get(hObject,'Value');
item_selected = items{index_selected};
display(item_selected);
```

When the end user selects an item in the list box, the callback function performs the following tasks:

- Gets all the items in the list box and stores them in the variable, items.
- Gets the numeric index of the selected item and stores it in the variable, index_selected.
- Gets the value of the selected item and stores it in the variable, item_selected.
- Displays the selected item in the MATLAB Command Window.

The example, “Interactive List Box App in GUIDE” on page 20-6 shows how to populate a list box with directory names.

Pop-Up Menu

Populate Items in the Pop-Up Menu

If you are developing an app using GUIDE, use the pop-up menu CreateFcn callback to add items to the pop-up menu.

This code is an example of a pop-up menu CreateFcn callback that populates the menu with the items, Red, Green, and Blue.

```
function popupmenu1_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns

% Hint: popupmenu controls usually have a white background on Windows.
if ispc && isequal(get(hObject,'BackgroundColor'),...
    get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
set(hObject,'String',{'Red';'Green';'Blue'});
```

The last line, `set(hObject,'String',{'Red';'Green';'Blue'})`, populates the contents of the pop-up menu.

Change the Selected Item

When the end user selects an item, the pop-up menu's `Value` property changes to a number that corresponds to the item's position in the menu. For example, a value of `1` corresponds to the first item in the list. If you want to change the selection in your code, then change the `Value` property to another number between `1` and the number of items in the menu.

For example, you can use the `handles` structure in GUIDE to access the pop-up menu and change the `Value` property:

```
set(handles.popupmenu1,'Value',2)
```

The first argument, `handles.popupmenu1`, might be different in your code, depending on the value of the pop-up menu `Tag` property.

Write the Callback Function

This code is an example of a pop-up menu callback function in GUIDE. Associate this function with the pop-up menu `Callback` property to make it execute when the end user selects an item from the menu.

```
function popupmenu1_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns contents...
%        contents{get(hObject,'Value')} returns selected item...
items = get(hObject,'String');
index_selected = get(hObject,'Value');
item_selected = items{index_selected};
display(item_selected);
```

When the user selects an item in the pop-up menu, the callback function performs the following tasks:

- Gets all the items in the pop-up menu and stores them in the variable, `items`.
- Gets the numeric index of the selected item and stores it in the variable, `index_selected`.
- Gets the value of the selected item and stores it in the variable, `item_selected`.
- Displays the selected item in the MATLAB Command Window.

Panel

Make the Panel Respond to Button Clicks

You can create a callback function that executes when the end user right-clicks or left-clicks on the panel. If you are working in GUIDE, then right-click the panel in the layout and select **View Callbacks** > **ButtonDownFcn** to create the callback function.

This code is an example of a `ButtonDownFcn` callback in GUIDE.

```
function uipanel1_ButtonDownFcn(hObject, eventdata, handles)
% hObject    handle to uipanel1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
display('Mouse button was pressed');
```

When the end user clicks on the panel, this function displays the text, `'Mouse button was pressed'`, in the Command Window.

Resize the Window and Panel

By default, GUIDE UIs cannot be resized, but you can override this behavior by selecting **Tools** > **GUI Options** and setting **Resize behavior** to **Proportional**.

When the UI window is resizable, the position of components in the window adjust as the user resizes it. If you have a panel in your UI, then the panel’s size will change with the window’s size. Use the panel’s `SizeChangedFcn` callback to make your app perform specific tasks when the panel resizes.

This code is an example of a panel’s `SizeChangedFcn` callback in a GUIDE app. When the user resizes the window, this function modifies the font size of static text inside the panel.

```
function uipanel1_SizeChangedFcn(hObject, eventdata, handles)
% hObject    handle to uipanel1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
set(hObject,'Units','Points')
panelSizePts = get(hObject,'Position');
panelHeight = panelSizePts(4);
set(hObject,'Units','normalized');
newFontSize = 10 * panelHeight / 115;
texth = findobj('Tag','text1');
set(texth,'FontSize',newFontSize);
```

If your UI contains nested panels, then they will resize from the inside-out (in child-to-parent order).

Note To make the text inside a panel resize automatically, set the `fontUnits` property to `'normalized'`.

Button Group

Button groups are similar to panels, but they also manage exclusive selection of radio buttons and toggle buttons. When a button group contains multiple radio buttons or toggle buttons, the button group allows the end user to select only one of them.

Do not code callbacks for the individual buttons that are inside a button group. Instead, use the button group's `SelectionChangedFcn` callback to respond when the end user selects a button.

This code is an example of a button group `SelectionChangedFcn` callback that manages two radio buttons and two toggle buttons.

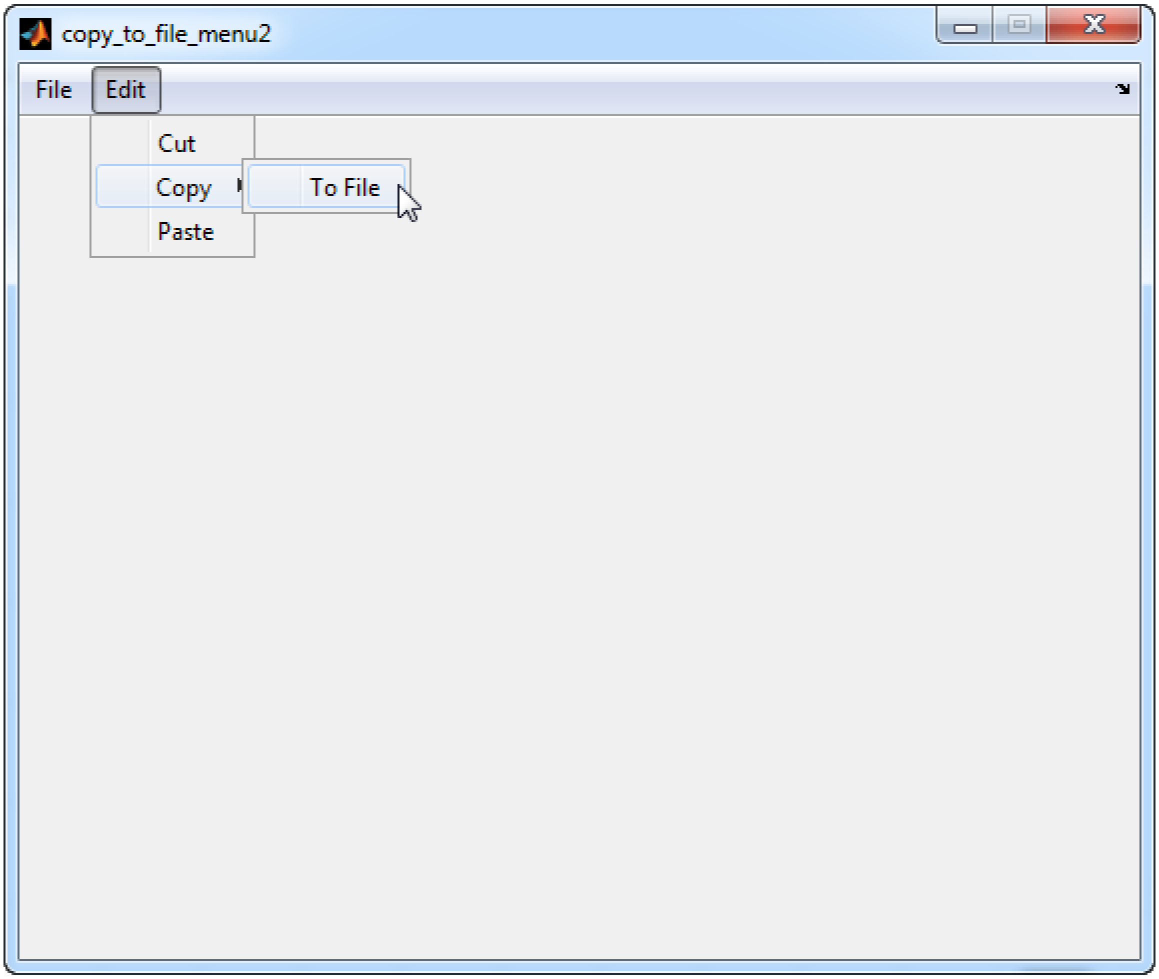
```
function uibuttongroup1_SelectionChangedFcn(hObject, eventdata, handles)
% hObject    handle to the selected object in uibuttongroup1
% eventdata  structure with the following fields
%    EventName: string 'SelectionChanged' (read only)
%    OldValue: handle of the previously selected object or empty
%    NewValue: handle of the currently selected object
% handles    structure with handles and user data (see GUIDATA)
switch get(eventdata.NewValue,'Tag') % Get Tag of selected object.
    case 'radiobutton1'
        display('Radio button 1');
    case 'radiobutton2'
        display('Radio button 2');
    case 'togglebutton1'
        display('Toggle button 1');
    case 'togglebutton2'
        display('Toggle button 2');
end
```

When the end user selects a radio button or toggle button in the button group, this function determines which button the user selected based on the button's `Tag` property. Then, it executes the code inside the appropriate `case`.

Note The button group's `SelectedObject` property contains a handle to the button that user selected. You can use this property elsewhere in your code to determine which button the user selected.

Menu Item

The code in this section contains example callback functions that respond when the end user selects **Edit > Copy > To File** in this menu.



```
% --------------------------------------------------------------------
function edit_menu_Callback(hObject, eventdata, handles)
% hObject    handle to edit_menu (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
display('Edit menu selected');

% --------------------------------------------------------------------
function copy_menu_item_Callback(hObject, eventdata, handles)
% hObject    handle to copy_menu_item (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
display('Copy menu item selected');

% --------------------------------------------------------------------
function tofile_menu_item_Callback(hObject, eventdata, handles)
% hObject    handle to tofile_menu_item (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
[filename,path] = uiputfile('myfile.m','Save file name');
```

The function names might be different in your code, depending on the tag names you specify in the GUIDE Menu Editor.

The callback functions trigger in response to these actions:

- When the end user selects the **Edit** menu, the `edit_menu_Callback` function displays the text, `'Edit menu selected'`, in the MATLAB Command Window.
- When the end user hovers the mouse over the **Copy** menu item, the `copy_menu_item_Callback` function displays the text, `'Copy menu item selected'`, in the MATLAB Command Window.
- When the end user clicks and releases the mouse button on the **To File** menu item, the `tofile_menu_item_Callback` function displays a dialog box that prompts the end user to select a destination folder and file name.

The `tofile_menu_item_Callback` function calls the `uiputfile` function to prompt the end user to supply a destination file and folder. If you want to create a menu item that prompts the user for an existing file, for example, if your UI has an **Open File** menu item, then use the `uigetfile` function.

When you create a cascading menu like this one, the intermediate menu items trigger when the mouse hovers over them. The final, terminating, menu item triggers when the mouse button releases over the menu item.

How to Update a Menu Item Check

You can add a check mark next to a menu item to indicate that an option is enabled. In GUIDE, you can select **Check mark this item** in the Menu Editor to make the menu item checked by default. Each time the end user selects the menu item, the callback function can turn the check on or off.

This code shows how to change the check mark next to a menu item.

```
if strcmp(get(hObject,'Checked'),'on')
    set(hObject,'Checked','off');
else
    set(hObject,'Checked','on');
end
```

The `strcmp` function compares two character vectors and returns `true` when they match. In this case, it returns `true` when the menu item's `Checked` property matches the character vector, `'on'`.

See "Create Menus for GUIDE Apps" on page 18-41 for more information about creating menu items in GUIDE.

Table

This code is an example of the table callback function, `CellSelectionCallback`. Associate this function with the table `CellSelectionCallback` property to make it execute when the end user selects cells in the table.

```
function uitable1_CellSelectionCallback(hObject, eventdata, handles)
% hObject    handle to uitable1 (see GCBO)
% eventdata  structure with the following fields
%   Indices: row and column indices of the cell(s) currently selected
% handles    structure with handles and user data (see GUIDATA)
data = get(hObject,'Data');
indices = eventdata.Indices;
r = indices(:,1);
c = indices(:,2);
```

```
linear_index = sub2ind(size(data),r,c);
selected_vals = data(linear_index);
selection_sum = sum(sum(selected_vals))
```

When the end user selects cells in the table, this function performs the following tasks:

- Gets all the values in the table and stores them in the variable, `data`.
- Gets the indices of the selected cells. These indices correspond to the rows and columns in `data`.
- Converts the row and column indices into linear indices. The linear indices allow you to select multiple elements in an array using one command.
- Gets the values that the end user selected and stores them in the variable, `selected_vals`.
- Sums all the selected values and displays the result in the Command Window.

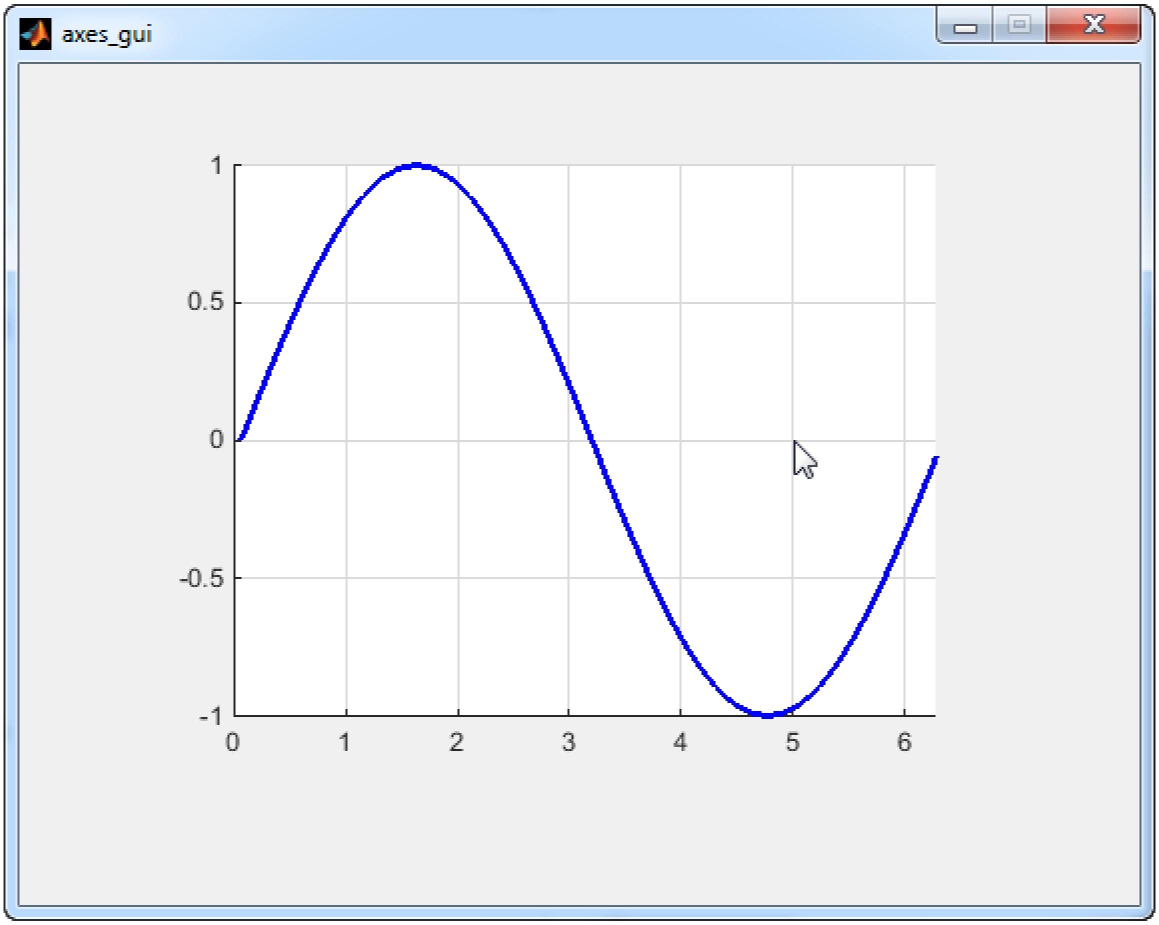
This code is an example of the table callback function, `CellEditCallback`. Associate this function with the table `CellEditCallback` property to make it execute when the end user edits a cell in the table.

```
function uitable1_CellEditCallback(hObject, eventdata, handles)
% hObject    handle to uitable1 (see GCBO)
% eventdata  structure with the following fields
%    Indices: row and column indices of the cell(s) edited
%    PreviousData: previous data for the cell(s) edited
%    EditData: string(s) entered by the user
%    NewData: EditData or its converted form set on the Data property.
% Empty if Data was not changed
% Error: error string when failed to convert EditData
data = get(hObject,'Data');
data_sum = sum(sum(data))
```

When the end user finishes editing a table cell, this function gets all the values in the table and calculates the sum of all the table values. The `ColumnEditable` property must be set to `true` in at least one column to allow the end user to edit cells in the table.

Axes

The code in this section is an example of an axes `ButtonDownFcn` that triggers when the end user clicks on the axes.



```
function axes1_ButtonDownFcn(hObject, eventdata, handles)
% hObject    handle to axes1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
pt = get(hObject,'CurrentPoint')
```

The coordinates of the pointer display in the MATLAB Command Window when the end user clicks on the axes (but not when that user clicks on another graphics object parented to the axes).

Note Most MATLAB plotting functions clear the axes and reset a number of axes properties, including the `ButtonDownFcn`, before plotting data. To create an interface that lets the end user plot data interactively, consider providing a component such as a push button to control plotting. Such components' properties are unaffected by the plotting functions. If you must use the axes `ButtonDownFcn` to plot data, then use functions such as `line`, `patch`, and `surface`.

See Also

Related Examples

- "Write Callbacks in GUIDE" on page 19-2
- "Create Callbacks for Apps Created Programmatically" on page 11-2

- “Callbacks in App Designer” on page 6-16

Examples of GUIDE UIs

GUIDE App With Parameters for Displaying Plots

Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

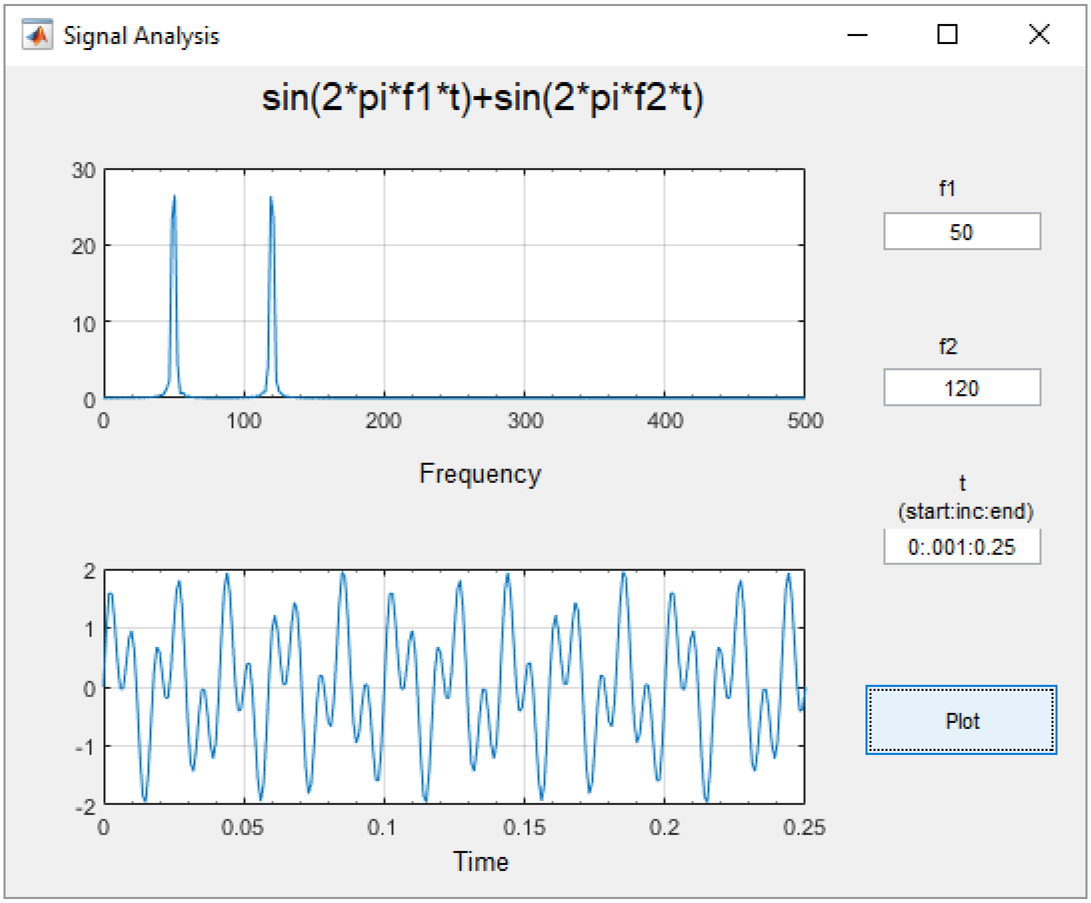
To continue editing an existing GUIDE app, see “GUIDE Migration Strategies” on page 3-7 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, “Develop Apps Using App Designer” instead.

This example shows how to examine and run a prebuilt GUIDE app. The app contains three edit fields and two axes. The axes display the frequency and time domain representations of a function that is the sum of two sine waves. The top two edit fields contain the frequency for each component sine wave. The third edit field contains the time range and sampling rate for the plots.

Open and Run the Example

Open and run the app. Change the default values in the **f1** and **f2** fields to change the frequency for each component sine wave. You can also change the three numbers (separated by colons) in the **t** field. The first and last numbers specify the window of time to sample the function. The middle number specifies the sampling rate.

Press the **Plot** button to see the graph of the function in the frequency and time domains.



Examine the Code

1. In GUIDE, click the **Editor** button to view the code.
2. Near the top of the Editor window, use the **Go To** button to navigate to the functions discussed below.

f1_input_Callback and f2_input_Callback

The `f1_input_Callback` function executes when the user changes the value in the **f1** edit field. The `f2_input_Callback` function responds to changes in the **f2** field, and it is almost identical to the `f1_input_Callback` function. Both functions check for valid user input. If the value in the edit field is invalid, the **Plot** button is disabled. Here is the code for the `f1_input_Callback` function.

```
f1 = str2double(get(hObject,'String'));
if isnan(f1) || ~isreal(f1)
    % Disable the Plot button and change its string to say why
    set(handles.plot_button,'String','Cannot plot f1');
    set(handles.plot_button,'Enable','off');
    % Give the edit text box focus so user can correct the error
    uicontrol(hObject);
else
    % Enable the Plot button with its original name
    set(handles.plot_button,'String','Plot');
    set(handles.plot_button,'Enable','on');
end
```

t_input_Callback

The `t_input_Callback` function executes when the user changes the value in the **t** edit field. This `try` block checks the value to make sure that it is numeric, that its length is between 2 and 1000, and that the vector is monotonically increasing.

```
try
    t = eval(get(handles.t_input,'String'));
    if ~isnumeric(t)
        % t is not a number
        set(handles.plot_button,'String','t is not numeric')
    elseif length(t) < 2
        % t is not a vector
        set(handles.plot_button,'String','t must be vector')
    elseif length(t) > 1000
        % t is too long a vector to plot clearly
        set(handles.plot_button,'String','t is too long')
    elseif min(diff(t)) < 0
        % t is not monotonically increasing
        set(handles.plot_button,'String','t must increase')
    else
        % Enable the Plot button with its original name
        set(handles.plot_button,'String','Plot')
        set(handles.plot_button,'Enable','on')
        return
    end

 catch EM
    % Cannot evaluate expression user typed
    set(handles.plot_button,'String','Cannot plot t');
    uicontrol(hObject);
end
```

The `catch` block changes the label on the **Plot** button to indicate that an input value was invalid. The `uicontrol` command sets the focus to the field that contains the erroneous value.

plot_button_Callback

The `plot_button_Callback` function executes when the user clicks the **Plot** button.

First, the callback gets the values in the three edit fields:

```
f1 = str2double(get(handles.f1_input,'String'));
f2 = str2double(get(handles.f2_input,'String'));
t = eval(get(handles.t_input,'String'));
```

Then callback uses values of `f1`, `f2`, and `t` to sample the function in the time domain and calculate the Fourier transform. Then, the two plots are updated:

```
% Create frequency plot in proper axes
plot(handles.frequency_axes,f,m(1:257));
set(handles.frequency_axes,'XMinorTick','on');
grid(handles.frequency_axes,'on');

% Create time plot in proper axes
```

```
plot(handles.time_axes,t,x);
set(handles.time_axes,'XMinorTick','on');
grid on
```

See Also

Related Examples

- “Write Callbacks in GUIDE” on page 19-2
- “Share Data Among Callbacks” on page 11-9

Interactive List Box App in GUIDE

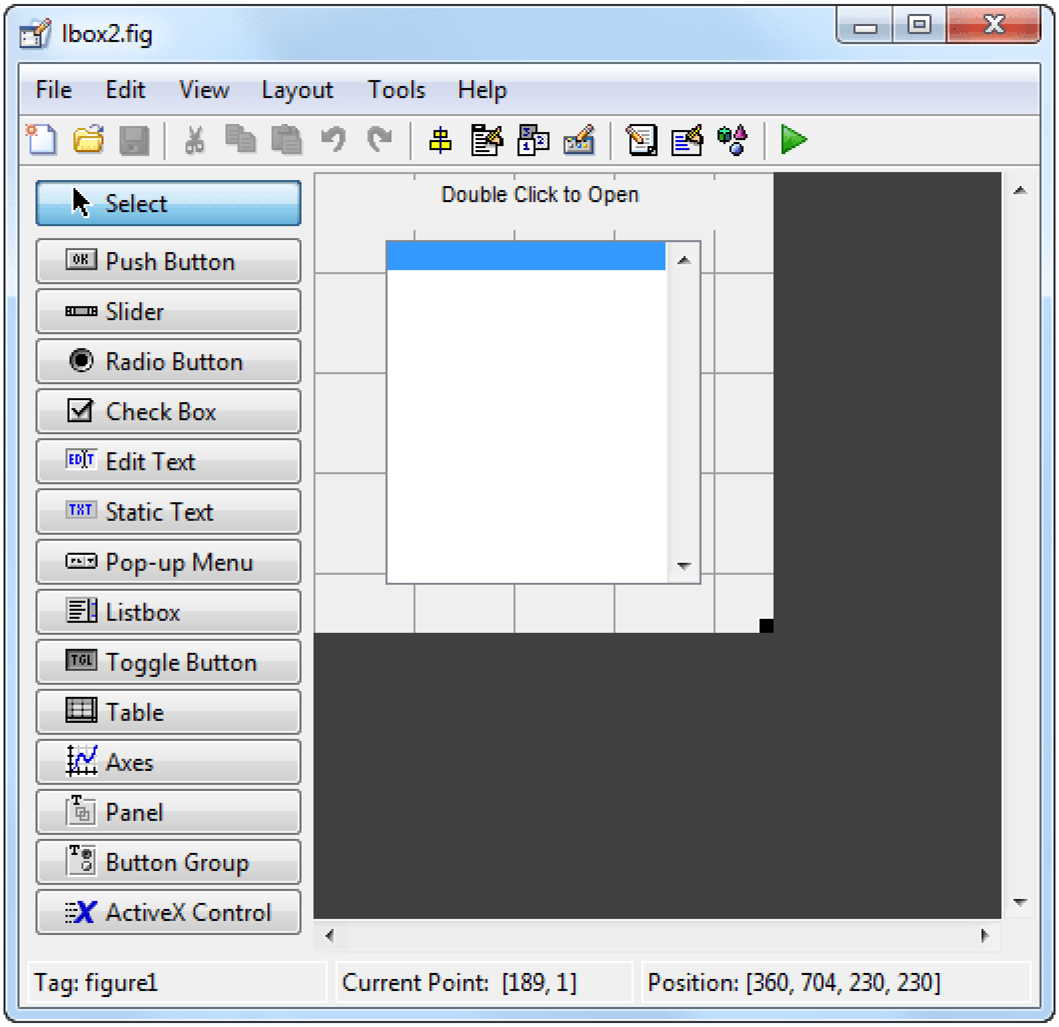
Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see "GUIDE Migration Strategies" on page 3-7 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, "Develop Apps Using App Designer" instead.

This example shows how to examine and run a prebuilt GUIDE app. The app contains a list box that displays the files in a particular folder. When you double-click an item in the list, MATLAB opens the item.

Open and Run the Example

Open the app in GUIDE, and click the **Run Figure** (green play button) to run it.



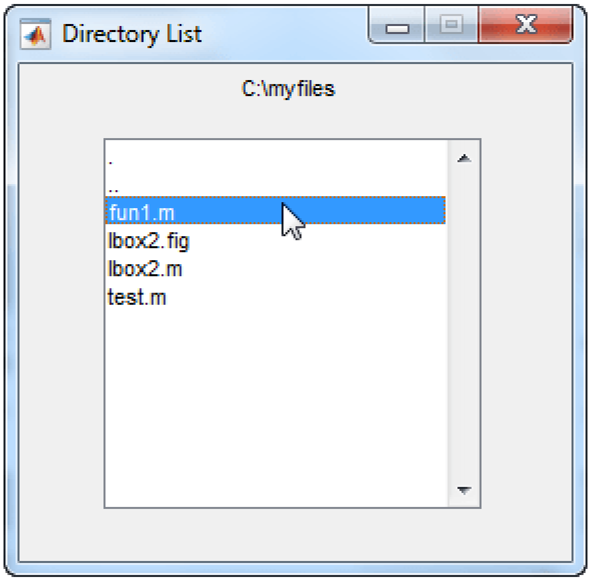
Tag: figure1

Current Point: [189, 1]

Position: [360, 704, 230, 230]

Alternatively, you can call the `lbox2` function in the Command Window with the `'dir'` name-value pair argument. The name-value pair argument allows you to list the contents of any folder. For example, this command lists the files in the C:\ folder on a Windows® system:

```
lbox2('dir','C:\')
```



Note: Before you can call `lbox2` in the Command Window, you must save the GUIDE files in a folder on your MATLAB® path. To save the files, select **File > Save As** in GUIDE.

Examine the Layout and Callback Code

1 In GUIDE, click the **Editor** button to view the code.

2 Near the top of the Editor window, use the **Go To** ▾ button to navigate to the functions discussed below.

lbox2_OpeningFcn

The callback function `lbox2_OpeningFcn` executes just before the list box appears in the UI for the first time. The following statements determine whether the user specified a path argument to the `lbox2` function.

```
if nargin == 3,
    initial_dir = pwd;
elseif nargin > 4
    if strcmpi(varargin{1},'dir')
        if exist(varargin{2},'dir')
            initial_dir = varargin{2};
        else
            errordlg('Input must be a valid directory','Input Argument Error!')
            return
        end
    else
        errordlg('Unrecognized input argument','Input Argument Error!');
        return;
    end
end
```

If nargin==3, then the only input arguments to lbox2_OpeningFcn are hObject, eventdata, and handles. Therefore, the user did not specify a path when they called lbox2, so the list box shows the contents of the current folder. If nargin>4, then the varargin input argument contains two additional items (suggesting that the user did specify a path). Thus, subsequent if statements check to see whether the path is valid.

listbox1_callback

The callback function listbox1_callback executes when the user clicks a list box item. This statement, near the beginning of the function, returns true whenever the user double-clicks an item in the list box:

```
if strcmp(get(handles.figure1,'SelectionType'),'open')
```

If that condition is true, then listbox1_callback determines which list box item the user selected:

```
index_selected = get(handles.listbox1,'Value');
file_list = get(handles.listbox1,'String');
filename = file_list{index_selected};
```

The rest of the code in this callback function determines how to open the selected item based on whether the item is a folder, FIG file, or another type of file:

```
    if  handles.is_dir(handles.sorted_index(index_selected))
        cd (filename)
        load_listbox(pwd,handles)
    else
        [path,name,ext] = fileparts(filename);
        switch ext
            case '.fig'
                guide (filename)
            otherwise
                try
                    open(filename)
                catch ex
                    errordlg(...
                      ex.getReport('basic'),'File Type Error','modal')
                end
        end
    end
```

See Also

Related Examples

- “Write Callbacks in GUIDE” on page 19-2
- “Share Data Among Callbacks” on page 11-9

Automatically Refresh Plot in a GUIDE App

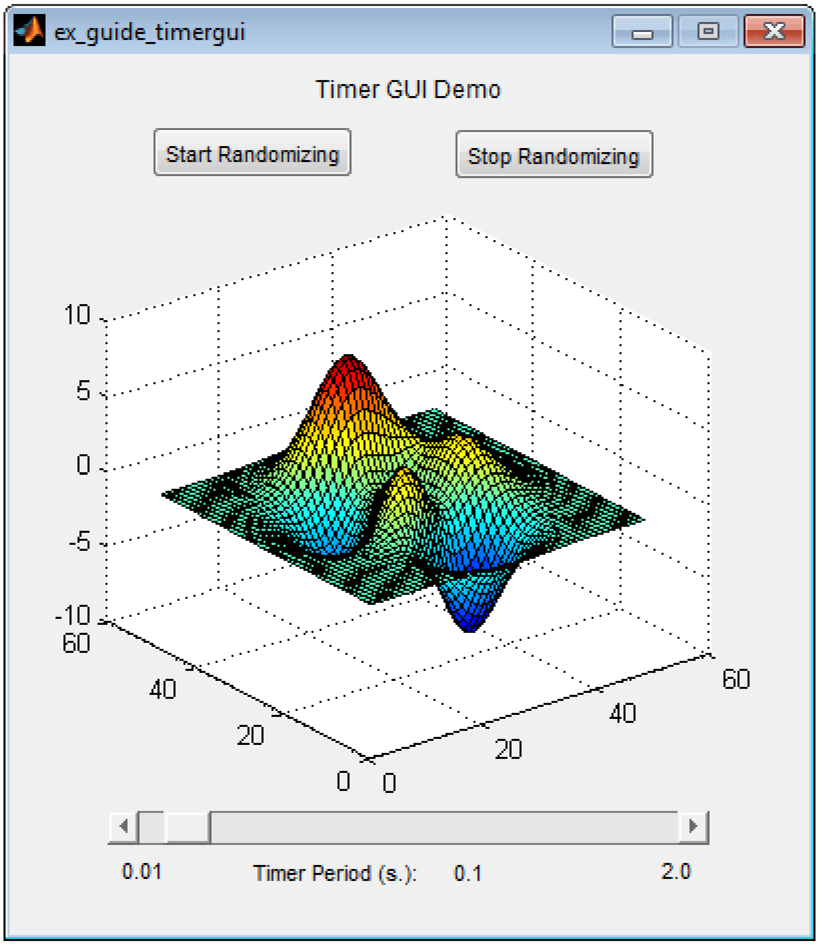
Note The GUIDE environment will be removed in a future release. After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but they will not be editable in GUIDE.

To continue editing an existing GUIDE app, see "GUIDE Migration Strategies" on page 3-7 for information on how to help maintain compatibility of the app with future MATLAB releases. To create new apps interactively, "Develop Apps Using App Designer" instead.

This example shows how to examine and run a prebuilt GUIDE app. The app displays a surface plot, adds random noise to the surface, and refreshes the plot at regular intervals. The app contains two buttons: one that starts adding random noise to the plot, and another that stops adding noise. The slider below the plot allows the user to set the refresh period between 0.01 and 2 seconds.

Open and Run the Example

Open and run the app. Move the slider to set the refresh interval between 0.01 and 2.0 seconds. Then click the **Start Randomizing** button to start adding random noise to the plotted function. Click the **Stop Randomizing** button to stop adding noise and refreshing the plot.



Examine the Code

1 In GUIDE, click the **Editor** button to view the code.

2 Near the top of the Editor window, use the **Go To** ▾ button to navigate to the functions discussed below.

ex_guide_timergui_OpeningFcn

The `ex_guide_timergui_OpeningFcn` function executes when the app opens and starts running. This command creates the `timer` object and stores it in the `handles` structure.

```
handles.timer = timer(...
    'ExecutionMode', 'fixedRate', ...       % Run timer repeatedly.
    'Period', 1, ...                        % Initial period is 1 sec.
    'TimerFcn', {@update_display,hObject}); % Specify callback function.
```

The callback function for the timer is `update_display`, which is defined as a local function.

update_display

The update_display function executes when the specified timer period elapses. The function gets the values in the ZData property of the Surface object and adds random noise to it. Then it updates the plot.

```
handles = guidata(hfigure);
Z = get(handles.surf,'ZData');
Z = Z + 0.1*randn(size(Z));
set(handles.surf,'ZData',Z);
```

periodsldr_Callback

The periodsldr_Callback function executes when the user moves the slider. It calculates the timer period by getting the slider value and truncating it. Then it updates the label below the slider and updates the period of the timer object.

```
% Read the slider value
period = get(handles.periodsldr,'Value');
% Truncate the value returned by the slider.
period = period - mod(period,.01);
% Set slider readout to show its value.
set(handles.slidervalue,'String',num2str(period))
% If timer is on, stop it, reset the period, and start it again.
if strcmp(get(handles.timer, 'Running'), 'on')
    stop(handles.timer);
    set(handles.timer,'Period',period)
    start(handles.timer)
else               % If timer is stopped, reset its period.
    set(handles.timer,'Period',period)
end
```

startbtn_Callback

The startbtn_Callback function calls the start method of the timer object if the timer is not already running.

```
if strcmp(get(handles.timer, 'Running'), 'off')
    start(handles.timer);
end
```

stopbtn_Callback

The stopbtn_Callback function calls the stop method of the timer object if the timer is currently running.

```
if strcmp(get(handles.timer, 'Running'), 'on')
    stop(handles.timer);
end
```

figure1_CloseRequestFcn

The figure1_CloseRequestFcn callback executes when the user closes the app. The function stops the timer object if it is running, deletes the timer object, and then deletes the figure window.

```
if strcmp(get(handles.timer, 'Running'), 'on')
    stop(handles.timer);
end
```

```
% Destroy timer
delete(handles.timer)
% Destroy figure
delete(hObject);
```

See Also

Related Examples

- “Timer Callback Functions”
- “Write Callbacks in GUIDE” on page 19-2

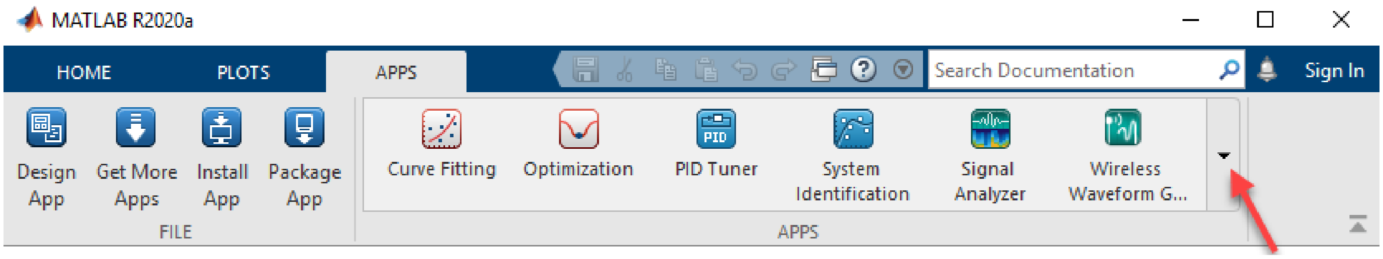
App Packaging

Packaging GUIs as Apps

Get and Create Apps

What Is an App?

A MATLAB app is a self-contained MATLAB program with a user interface that automates a task or calculation. All the operations required to complete the task — getting data into the app, performing calculations on the data, and displaying results are performed within the app. Apps are included in many MATLAB products. In addition, you can design your own apps using the App Designer development environment. The **Apps** tab on the MATLAB Toolstrip displays all currently installed apps when you click the drop-down arrow on the far right of the toolstrip.



Note You cannot run MATLAB apps using the MATLAB Runtime. Apps are for MATLAB to MATLAB deployment. To run code using the MATLAB Runtime, the code must be packaged using MATLAB Compiler.

Where to Get Apps

There are three key ways to get apps:

- MATLAB Products

  Many MATLAB products, such as Curve Fitting Toolbox™, Signal Processing Toolbox™, and Control System Toolbox™ include apps. In the apps gallery, you can see the apps that come with your installed products.
- Create Your Own

  App Designer is the recommended environment for building apps in MATLAB. You can create your own MATLAB app and package it into a single file that you can distribute to others. The apps packaging tool automatically finds and includes all the files needed for your app. It also identifies any MATLAB products required to run your app.

  You can share your app directly with other users, or share it with the MATLAB user community by uploading it to the MATLAB File Exchange. When others install your app, they do not need to be concerned with the MATLAB search path or other installation details.

  Watch this video for an introduction to creating apps:

  Packaging and Installing MATLAB Apps (2 min, 58 sec)
- Add-Ons

  Apps (and other files) uploaded to the MATLAB File Exchange are available from within MATLAB:

1. On the **Home** tab, in the **Environment** section, click the **Add-Ons** arrow button.
2. Click **Get Add-Ons**.
3. Search for apps by name or descriptive text.

Why Create an App?

When you create an app package, MATLAB creates a single app installation file (`.mlappinstall`) that enables you and others to install your app easily.

In particular, when you package an app, the app packaging tool:

- Performs a dependency analysis that helps you find and add the files your app requires.
- Reminds you to add shared resources and helper files.
- Stores information you provide about your app with the app package. This information includes a description, a list of additional MATLAB products required by your app, and a list of supported platforms.
- Automates app updates (versioning).

In addition when others install your app:

- It is a one-click installation.
- Users do not need to manage the MATLAB search path or other installation details.
- Your app appears alongside MATLAB toolbox apps in the apps gallery.

Best Practices and Requirements for Creating an App

Best practices:

- Write the app as an interactive application with a user interface written in the MATLAB language.
- All interaction with the app is through the user interface.
- Make the app reusable. Do not make it necessary for a user to restart the app to use different data or inputs with it.
- Ensure the main function returns the handle of the main figure. (The main function created by GUIDE returns the figure handle by default.)

  Although not a requirement, doing so enables MATLAB to remove the app files from the search path when users exit the app.
- If you want to share your app on MATLAB File Exchange, you must release it under a BSD license. In addition, there are restrictions on the use of binary files such as MEX-files, p-coded files, or DLLs.

Requirements:

- The main file must be a function (not a script).
- Because you invoke apps by clicking an icon in the apps gallery, the main function cannot have any required input arguments. However, you can define optional input arguments. One way to define optional input arguments is by using `varargin`.

See Also

Related Examples

- “Package Apps From the MATLAB Toolstrip” on page 21-5
- “Modify Apps” on page 21-9
- “Ways to Share Apps” on page 21-10

Package Apps From the MATLAB Toolstrip

You can package any MATLAB app you create into a single file that can be easily shared with others. When you package an app, MATLAB creates a single app installation file (`.mlappinstall`). The installation file enables you and others to install your app and access it from the apps gallery without concern for installation details or the MATLAB path.

Note As you enter information in the Package Apps dialog box, MATLAB creates and saves a `.prj` file continuously. A `.prj` file contains information about your app, such as included files and a description. Therefore, if you exit the dialog box before clicking the **Package** button, the `.prj` file remains, even though a `.mlappinstall` file is not created. The `.prj` file enables you to quit and resume the app creation process where you left off.

To create an app installation file:

1. On the desktop Toolstrip, on the **Home** tab, click the **Add-Ons** down-arrow.
2. Click **Package App**.
3. In the Package App dialog box, click **Add main file** and specify the file that you use to run the app you created.

   The main file must be callable with no input and must be a function or method, not a script. MATLAB analyzes the main file to determine if there are other files used in the app. For more information, see “App Packaging Dependency Analysis” on page 21-16.

   Tip The main file must return the figure handle of your app for MATLAB to remove your app files from the search path when users exit the app. For more information, see “What Is the MATLAB Search Path?”

   (Functions created by GUIDE return the figure handle.)
4. If your app requires additional files that are not listed under **Files included through analysis**, add them by clicking **Add files/folders**.

   You can include external interfaces, such as MEX-files or Java® in the `.mlappinstall` file, although doing so can restrict the systems on which your app can run.
5. Describe your app.

   a. In the **App Name** field, type an app name.

      If you install the app, MATLAB uses the name for the `.mlappinstall` file and to label your app in the apps gallery.

   b. Optionally, specify an app icon.

      Click the icon to the left of the **App Name** field to select an icon for your app or to specify a custom icon. MATLAB automatically scales the icon for use in the Install dialog box, App gallery, and quick access toolbar.

   c. Optionally, select a previously saved screenshot to represent your app.

   d. Optionally, specify author information.

   e. In the **Description** field, describe your app so others can decide if they want to install it.

f. Identify the products on which your app depends.

   Click the plus button on the right side of the **Products** field, select the products on which your app depends, and then click **Apply Changes**. Keep in mind that your users must have all of the dependent products installed on their systems.

After you create the package, when you select a `.mlappinstall` file in the Current Folder browser, MATLAB displays the information you provided (except your email address and company name) in the Current Folder browser **Details** panel. If you share your app in the MATLAB Central File Exchange, the same information also displays there. The screenshot you select, if any, represents your app in File Exchange.

6. Click **Package**.

   As part of the app packaging process, MATLAB creates a `.prj` file that contains information about your app, such as included files and a description. The `.prj` file enables you to update the files in your app without requiring you to respecify descriptive information about the app.

7. In the Build dialog box, note the location of the installation file (`.mlappinstall`), and then click **Close**.

For information on installing the app, see “Install Add-Ons from File”.

See Also

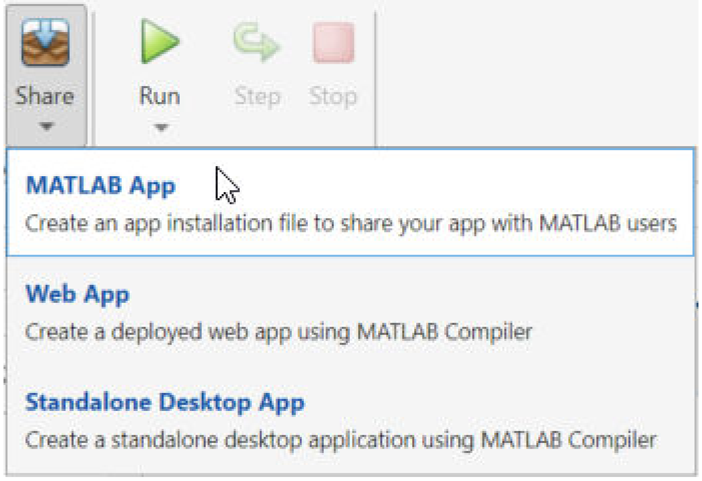
Related Examples

- “Modify Apps” on page 21-9
- “Ways to Share Apps” on page 21-10
- “MATLAB App Installer File — mlappinstall” on page 21-15
- “App Packaging Dependency Analysis” on page 21-16

Package Apps in App Designer

After creating an app in App Designer, you can package it into a single installer file that you can easily share with others. The underlying functionality for packaging apps in App Designer is the same as the functionality that underlies the **Add-Ons** > **Package App** option in the MATLAB Toolstrip.

1 In App Designer, select the **Designer** tab. Then select **Share** > **MATLAB App**.

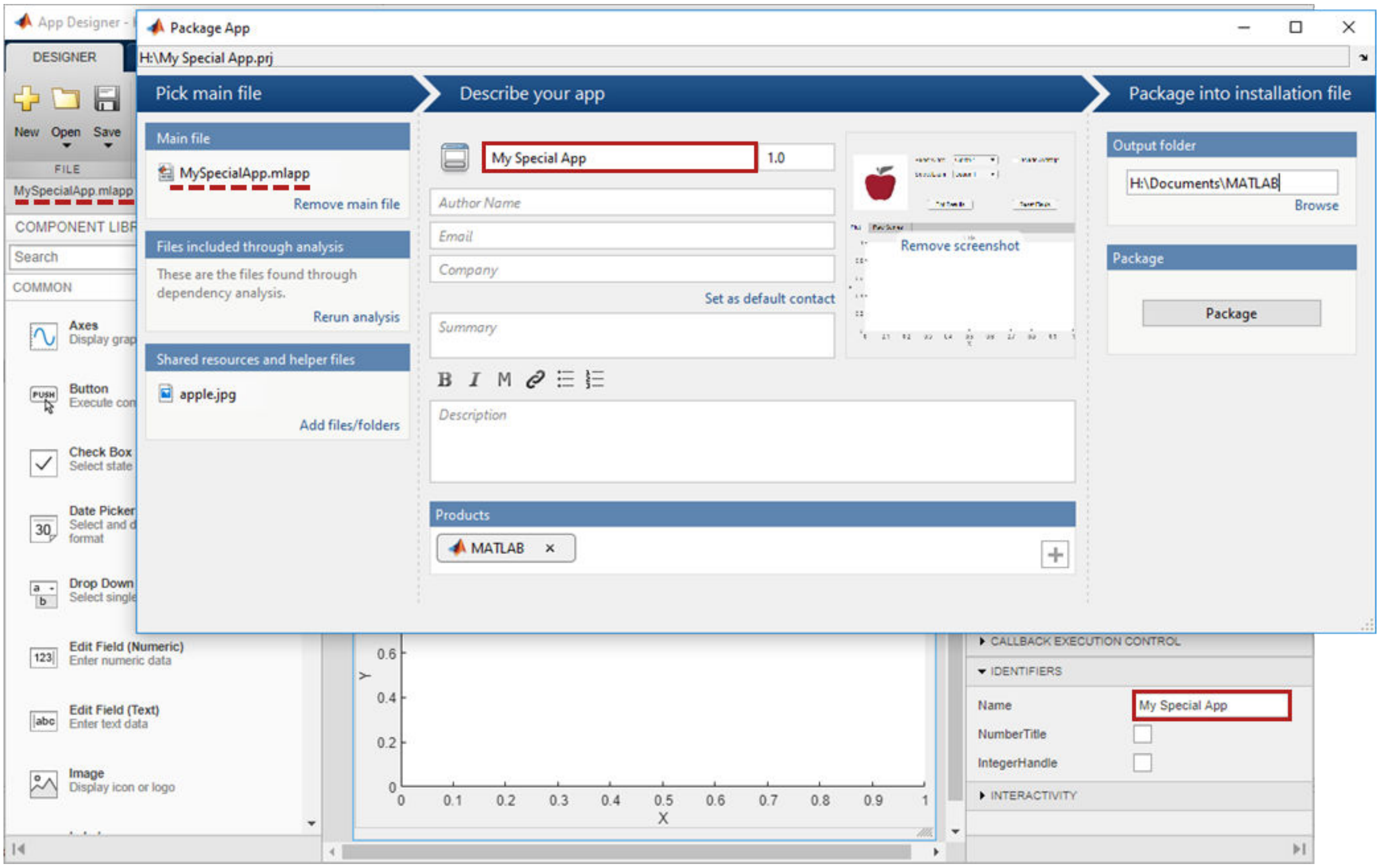


Share
Run
Step
Stop
MATLAB App
Create an app installation file to share your app with MATLAB users
Web App
Create a deployed web app using MATLAB Compiler
Standalone Desktop App
Create a standalone desktop application using MATLAB Compiler

MATLAB opens the Package App dialog box.

2 The Package App dialog box has the following items pre-populated:

- The application name matches the name assigned to the figure in App Designer.
- The **Main file** is the MLAPP file you currently have selected for editing.
- The **Output folder** is the folder location where the installation file will be saved.
- The files listed under **Files included through analysis** include any files MATLAB detected as dependent files. You can add additional files by clicking **Add files/folders** under **Shared resources and helper files**.



H:\My Special App.prj

Main file: MySpecialApp.mlapp

Shared resources and helper files: apple.jpg

My Special App 1.0

Products: MATLAB

Output folder: H:\Documents\MATLAB

Name: My Special App

NumberTitle □

IntegerHandle □

3 Specify details to display in the apps gallery. Enter the appropriate information in these fields: **Author Name**, **Email**, **Company**, **Summary**, and **Description**.

4 In the **Products** section, select the products that are required to run the app. Keep in mind that your users must have all of the dependent products installed on their systems.

5 Click **Select screenshot** to specify an icon to display in the apps gallery.

6 Click **Package** to create the `.mlappinstall` file to share with your users. Later, if you click the **Package App** button in the App Designer Toolstrip again, the Package App dialog box opens the most recently modified `.prj` file for the MLAPP file.

See Also

Related Examples

- “Package Apps From the MATLAB Toolstrip” on page 21-5
- “Ways to Share Apps” on page 21-10
- “MATLAB App Installer File — mlappinstall” on page 21-15
- “App Packaging Dependency Analysis” on page 21-16

Modify Apps

When you update the files included in a `.mlappinstall` file, you recreate and overwrite the original app. You cannot maintain two versions of the same app.

To update files in an app you created:

1. In the Current Folder browser, navigate to the folder containing the project file (`.prj`) that MATLAB created when you packaged the app.

   By default, MATLAB writes the `.prj` file to the folder that was the current folder when you packaged the app.
2. From the Current Folder browser, double-click the project file for your app package, *`appname`*`.prj`.

   The Package App dialog box opens.
3. Adjust the information in the dialog box to reflect your changes by doing any or all of the following:

   - If you made code changes, add the main file again, and refresh the files included through analysis.
   - If your code calls additional files that are not included through analysis, add them.
   - If you want anyone who installs your app over a previous installation to be informed that the content is different, change the version.

     Version numbers must be a combination of integers and periods, and can include up to three periods — `2.3.5.2`, for example.

     Anyone who attempts to install a revision of your app over another version is notified that the version number is changed. The user can continue or cancel the installation.
   - If your changes introduce different product dependencies, adjust the product list in the **Products** field. Keep in mind that your users must have all of the dependent products installed on their systems.
4. Click **Package**.

See Also

Related Examples

- “Ways to Share Apps” on page 21-10
- “MATLAB App Installer File — mlappinstall” on page 21-15
- “App Packaging Dependency Analysis” on page 21-16

Ways to Share Apps

There are several ways to share your apps.

- “Share MATLAB Files Directly” on page 21-10 — This approach is the simplest way to share an app, but your users must have MATLAB installed on their systems, as well as other MathWorks products that your app depends on. They must also be familiar with executing commands in the MATLAB Command Window and know how to manage the MATLAB path.
- “Package Your App” on page 21-12 — This approach uses the app packaging tool provided with MATLAB. When your users install a packaged app, the app appears in the **Apps** tab in the MATLAB Toolstrip. This approach is useful for sharing apps with larger audiences, or when your users are less familiar with executing commands in the MATLAB Command Window or managing the MATLAB path. As in the case of sharing MATLAB files directly, your users must have MATLAB installed on their systems (as well as other MathWorks products that your app depends on).
- “Create a Deployed Web App” on page 21-13 — This approach lets you create apps that users within an organization can run in their web browsers. To deploy a web app, you must have MATLAB Compiler installed on your system. Your users must have a web browser installed that can access your intranet, but they do not need to have MATLAB installed.
- “Create a Standalone Desktop Application” on page 21-13 — This approach lets you share desktop apps with users that do not have MATLAB installed on their systems. To create the standalone application, you must have MATLAB Compiler installed on your system. To run the application, your users must have MATLAB Runtime installed on their systems. For more information, see https://www.mathworks.com/products/compiler/matlab-runtime.html.

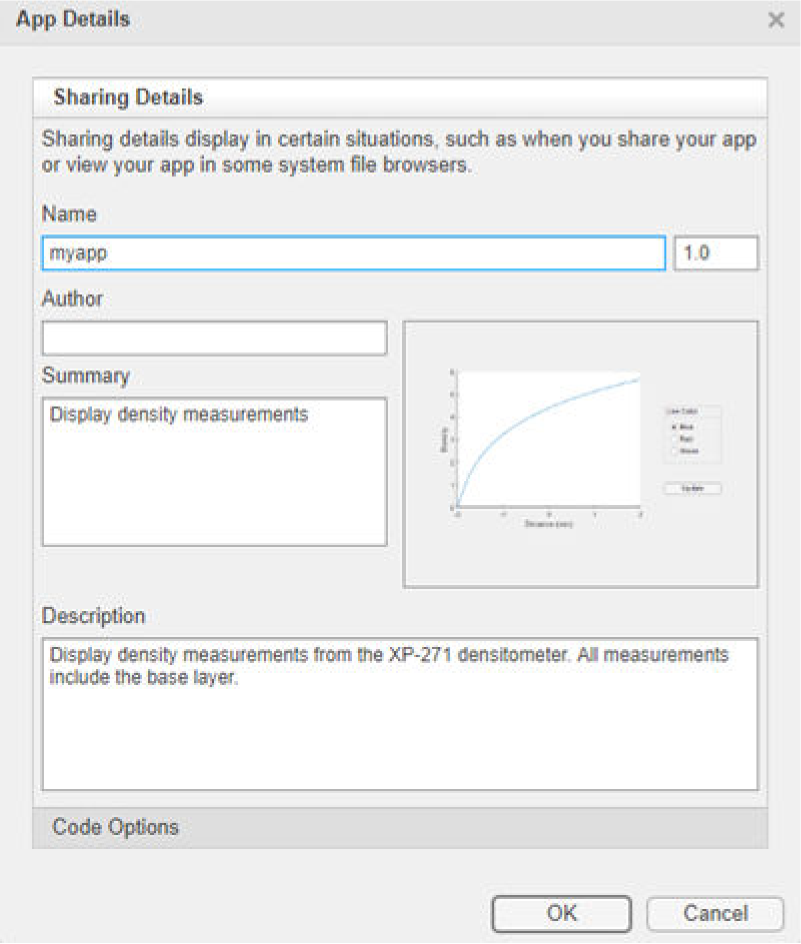
Share MATLAB Files Directly

If you created your app in GUIDE, share the `.fig` file, the `.m` file, and all other dependent files with your users.

If you created your app programmatically, share all `.m` files and other dependent files with your users.

If you created your app in App Designer, share the `.mlapp` file and all other dependent files with your users. To provide a richer file browsing experience for your users, provide a name, version, author, summary, and description by clicking **App Details** ☰ in the **Designer** tab of the App Designer toolstrip. The **App Details** dialog box also provides an option for specifying a screen shot. If you do not specify a screen shot, App Designer captures and updates a screen shot automatically when you run the app.

MATLAB provides your app details to some operating systems for display in their file browsers. Specifying apps details also makes it easier to package and compile your apps. The `.mlapp` file provides those details automatically to those interfaces.



App Details

Sharing Details

Sharing details display in certain situations, such as when you share your app or view your app in some system file browsers.

Name: myapp 1.0

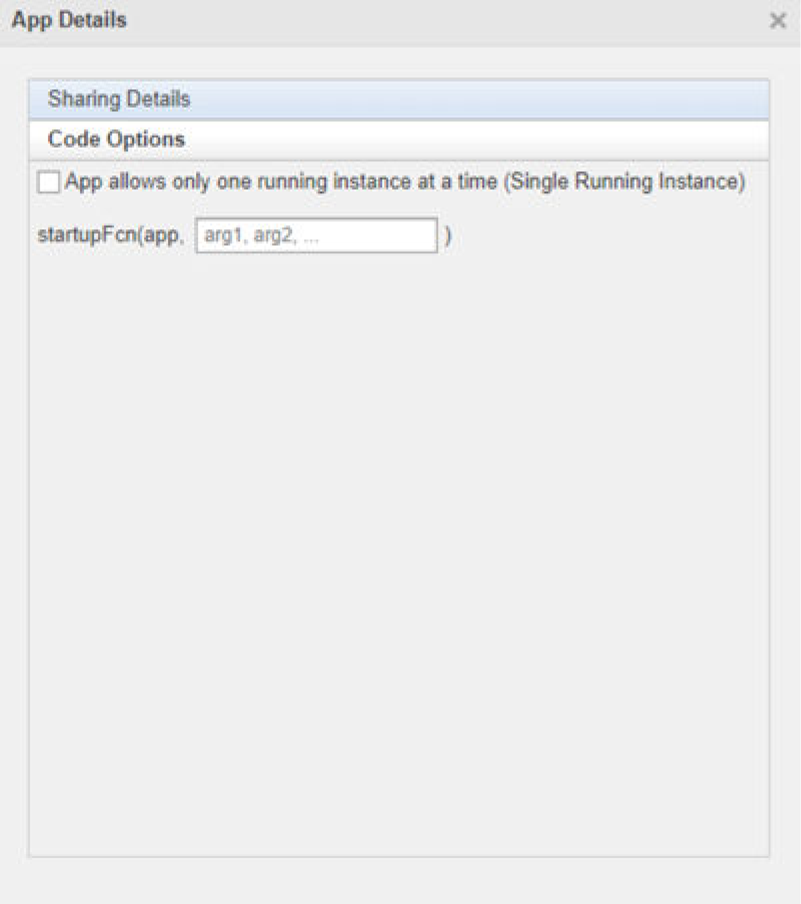
Author:

Summary: Display density measurements

Description: Display density measurements from the XP-271 densitometer. All measurements include the base layer.

Code Options

To specify input arguments and whether your app can run multiple instances at a time or only a single instance, expand the **Code Options** section and select from the available options.



□ App allows only one running instance at a time (Single Running Instance)

startupFcn(app, arg1, arg2, ...)

Package Your App

To package your app and make it accessible in the MATLAB **Apps** tab, create an `.mlappinstall` file by following the steps in “Package Apps in App Designer” on page 21-7 or “Package Apps From the MATLAB Toolstrip” on page 21-5. The resulting `.mlappinstall` file includes all dependent files.

You can share the `.mlappinstall` file directly with your users. To install it, they must double-click the `.mlappinstall` file in the MATLAB **Current Folder** browser.

Alternatively, you can share your app as an add-on by uploading the `.mlappinstall` file to MATLAB Central File Exchange. Your users can find and install your add-on from the MATLAB Toolstrip by performing these steps:

1. In the MATLAB Toolstrip, on the **Home** tab, in the **Environment** section, click the **Add-Ons** icon.
2. Find the add-on by browsing through available categories on the left side of the Add-On Explorer window. Use the search bar to search for an add-on using a keyword.
3. Click the add-on to open its detailed information page.
4. On the information page, click **Add** to install the add-on.

Note Although `.mlappinstall` files can contain any files you specify, MATLAB Central File Exchange places additional limitations on submissions. Your app cannot be submitted to File Exchange when it contains any of the following files:

- MEX-files
- Other binary executable files, such as DLLs. (Data and image files are typically acceptable.)

Create a Deployed Web App

Web apps are MATLAB apps that can run in a web browser. You create an interactive MATLAB app using App Designer, package it using MATLAB Compiler, and host it using either the development version of MATLAB Web App Server™ in MATLAB Compiler or the MATLAB Web App Server product. Each web app has a unique URL and can be accessed from a web browser using HTTP or HTTPS protocols. The server has a home page listing all available hosted web apps. You share web apps by sharing the unique URL to a web app or the URL to the home page of the server.

Creating web apps requires MATLAB Compiler, and only apps designed using App Designer can be deployed as web apps. In addition, certain functionality is not supported in deployed web apps. For more information, see "Web App Limitations and Unsupported Functionality" (MATLAB Compiler).

Once you have MATLAB Compiler on your system, package your MATLAB app into a web app from within App Designer by clicking **Share** in the **Designer** tab and selecting **Web App**. You can deploy your web app directly to the server by specifying the server URL in the packaging dialog. The format of the server URL is: `https://`*`webAppServer`*`:`*`PortNumber`*`/webapps/home/index.html`.

The ability to directly upload your web app to a server is only supported in the MATLAB Web App Server product and requires authentication to be enabled. For details, see "Authentication" (MATLAB Web App Server).

For more information on web apps, see "Web Apps" (MATLAB Compiler).

Create a Standalone Desktop Application

Creating a standalone desktop application lets you share an app with users who do not have MATLAB on their systems. However, you must have MATLAB Compiler installed on your system to create the standalone application. Your users must have MATLAB Runtime on their systems to run the app.

Once you have MATLAB Compiler on your system, you can open the Application Compiler from within App Designer by clicking **Share** in the **Designer** tab and selecting **Standalone Desktop App**.

If you used GUIDE or created your app programmatically, you can open the Application Compiler from the MATLAB Toolstrip, on the **Apps** tab, by clicking the **Application Compiler** icon.

See "Create Standalone Application from MATLAB Function" (MATLAB Compiler) for instructions on using the Application Compiler.

See Also

Related Examples

- "Get and Create Apps" on page 21-2
- "Ways to Build Apps" on page 1-2

MATLAB App Installer File — mlappinstall

A MATLAB app installer file, `.mlappinstall`, is an archive file for sharing an app you created using MATLAB. A single app installer file contains everything necessary to install and run an app: the source code, supporting data, information (such as product dependencies), and the app icon.

An `.mlappinstall` file is a compressed package that conforms to the Open Packaging Conventions (OPC) interoperability standard. You can search for and install `.mlappinstall` files using your operating system file browser. When you select an `.mlappinstall` file in Windows Explorer or Quick Look (Mac OS), the browser displays properties for the file, such as `Authors` and `Release`. Use these properties to search for `.mlappinstall` files. Use the `Tags` property to add custom searchable text to the file.

See Also

Related Examples

- “Package Apps From the MATLAB Toolstrip” on page 21-5

App Packaging Dependency Analysis

When you create an app package, MATLAB analyzes your main file and attempts to include all the files that your app uses. However, MATLAB is not guaranteed to find every dependent file. It does not find files for functions that your code references as character vectors (for instance, as arguments to `eval`, `feval`, and callback functions). In addition, MATLAB can include some files that the main file never calls when it runs.

Dependency analysis searches for the following types of files:

- Executable files, such as MATLAB program files, P-files, Fig-files, and MEX-files.
- Files that your app accesses by calling standard and low-level I/O functions. These dependent files include text files, spreadsheets, images, audio, video, and XML files.
- Files that your app accesses by calling any of these functions: `audioinfo`, `audioread`, `csvread`, `daqread`, `dlmread`, `fileread`, `fopen`, `imfinfo`, `importdata`, `imread`, `load`, `matfile`, `mmfileinfo`, `open`, `readtable`, `type`, `VideoReader`, `xlsfinfo`, `xlsread`, `xmlread`, and `xslt`.

Dependency analysis does not search for Java classes, `.jar` files, or files stored in a scientific format such as NetCDF or HDF. Click **Add files/folders** in the Package Apps dialog box to add these types of files manually.

See Also

`matlab.codetools.requiredFilesAndProducts`