

# MATLAB® Production Server™

Code Deployment



# MATLAB®

R2021a



## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*MATLAB® Production Server™ Code Deployment*

© COPYRIGHT 2012–2021 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

### Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

### Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

### Revision History

March 2014	Online only	New for Version 1.2 (Release R2014a)
October 2014	Online only	Revised for Version 2.0 (Release R2014b)
March 2015	Online only	Revised for Version 2.1 (Release R2015a)
September 2015	Online only	Revised for Version 2.2 (Release R2015b)
March 2016	Online only	Revised for Version 2.3 (Release 2016a)
September 2016	Online only	Revised for Version 2.4 (Release 2016b)
March 2017	Online only	Revised for Version 3.0 (Release 2017a)
September 2017	Online only	Revised for Version 3.0.1 (Release R2017b)
March 2018	Online only	Revised for Version 3.1 (Release R2018a)
September 2018	Online only	Revised for Version 4.0 (Release R2018b)
March 2019	Online only	Revised for Version 4.1 (Release R2019a)
September 2019	Online only	Revised for Version 4.2 (Release R2019b)
March 2020	Online only	Revised for Version 4.3 (Release R2020a)
September 2020	Online only	Revised for Version 4.4 (Release R2020b)
March 2021	Online only	Revised for Version 4.5 (Release R2021a)

## Write Deployable MATLAB Code

**1**

<b>MATLAB Coding Guidelines</b> .....	<b>1-2</b>
<b>State-Dependent Functions</b> .....	<b>1-3</b>
Does My MATLAB Function Carry State? .....	<b>1-3</b>
Defensive Coding Practices .....	<b>1-3</b>
Techniques for Preserving State .....	<b>1-4</b>
<b>Deploying MATLAB Functions Containing MEX Files</b> .....	<b>1-5</b>
<b>Supported MATLAB Data Types for Client and Server Marshaling</b> .....	<b>1-6</b>
Supported Data Types .....	<b>1-6</b>
Partially Supported Data Types .....	<b>1-6</b>
Unsupported Data Types .....	<b>1-6</b>

## Create a Deployable Archive from MATLAB Production Server Code

**2**

<b>Create Deployable Archive for MATLAB Production Server</b> .....	<b>2-2</b>
Create MATLAB Function .....	<b>2-2</b>
Create Deployable Archive with Production Server Compiler App .....	<b>2-2</b>
Customize Application and Its Appearance .....	<b>2-3</b>
Package Application .....	<b>2-3</b>
<b>Package Deployable Archives with Production Server Compiler App</b> .....	<b>2-5</b>
Create Function In MATLAB .....	<b>2-5</b>
Create Deployable Archive with Production Server Compiler App .....	<b>2-5</b>
Customize the Application and Its Appearance .....	<b>2-6</b>
Package the Application .....	<b>2-6</b>
<b>Package Deployable Archives from Command Line</b> .....	<b>2-8</b>
Execute Compiler Projects with deploytool .....	<b>2-8</b>
Package a Deployable Archive with mcc .....	<b>2-8</b>
Differences Between Compiler Apps and Command Line .....	<b>2-8</b>
<b>Modifying Deployed Functions</b> .....	<b>2-10</b>

3

<b>Customize an Application</b> .....	3-2
Customize the Installer .....	3-2
Manage Required Files in Compiler Project .....	3-4
Sample Driver File Creation .....	3-5
Specify Files to Install with Application .....	3-6
Additional Runtime Settings .....	3-6
<b>Manage Support Packages</b> .....	3-7
Using a Compiler App .....	3-7
Using the Command Line .....	3-7

Advanced Uses of the Command Line Compiler

4

<b>Simplify Compilation Using Macros</b> .....	4-2
Macros .....	4-2
Working With Macros .....	4-2
<b>Invoke MATLAB Build Options</b> .....	4-4
Specify Full Path Names to Build MATLAB Code .....	4-4
Using Bundles to Build MATLAB Code .....	4-4
<b>MATLAB Runtime Component Cache and Deployable Archive Embedding</b> .....	4-6
Overriding Default Behavior .....	4-7
For More Information .....	4-7

Functions

5

Apps

6

Persistence

7

<b>Use a Data Cache to Persist Data</b> .....	7-2
Example: Increment a Counter Using a Data Cache .....	7-3

<b>Example: Calculate the Shortest Route Between Cities Using Persistence</b>	
.....	7-5
Step 1: Write MATLAB Code that uses Persistence Functions	7-5
Step 2: Run Example in Testing Workflow	7-9
Step 3: Run Example in Deployment Workflow	7-10

## Persistence Functions

8

## MATLAB Client

9

<b>Connect MATLAB Session to MATLAB Production Server</b>	9-2
When to Use MATLAB Client for MATLAB Production Server	9-2
Install MATLAB Client for MATLAB Production Server	9-2
Connect MATLAB Session to MATLAB Production Server	9-2
System Requirements	9-3
Synchronous Function Execution	9-3
Supported Data Types	9-3
<b>Execute Deployed MATLAB Functions</b>	9-5
Install MATLAB Client for MATLAB Production Server	9-5
Deploy MATLAB Function on Server	9-5
Install MATLAB Production Server Add-On for the Deployable Archive	9-6
Manage Installed Add-On	9-8
Invoke Deployed MATLAB Function	9-9
<b>Configure Client-Server Communication</b>	9-11
Configure Timeouts and Retries	9-11
Update Server Configuration	9-12
<b>Application Access Control</b>	9-14
Prerequisites	9-14
Configure Access Control	9-14
<b>Execute Deployed Functions Using HTTPS</b>	9-17
Save SSL Certificate of Server	9-17
Install Add-On Using HTTPS	9-18
Manage Default Protocol for Client-Server Communication	9-18
<b>Manage Add-Ons</b>	9-20
Install Add-Ons	9-20
Remove Add-Ons	9-22
Get Information about Add-Ons	9-22
Manage Add-Ons	9-23
Manage Access to Applications Deployed on Server	9-23
<b>Deploy Add-Ons</b>	9-25
Prerequisites	9-26

Create Standalone Executables That Use Add-Ons . . . . .	9-26
Create Shared Libraries or Software Components That Use Add-Ons . . .	9-27
Create Deployable Archives That Use Add-Ons . . . . .	9-28

**MATLAB Client Functions**

---

**10**

**MATLAB Client Apps**

---

**11**

# Write Deployable MATLAB Code

---

- “MATLAB Coding Guidelines” on page 1-2
- “State-Dependent Functions” on page 1-3
- “Deploying MATLAB Functions Containing MEX Files” on page 1-5
- “Supported MATLAB Data Types for Client and Server Marshaling” on page 1-6

## MATLAB Coding Guidelines

When writing MATLAB code for deployment to MATLAB Production Server you must adhere to the same guidelines as when writing code for deployment with MATLAB Compiler™ or MATLAB Compiler SDK™. In addition, code deployed to MATLAB Production Server must adhere to additional guidelines:

- functions cannot depend on nor change MATLAB state.

Functions deployed with MATLAB Production Server may not always execute on the same instance of the MATLAB Runtime. Each worker access a different MATLAB Runtime instance.

- explicitly use `varargin` and `varargout` for functions with variable inputs and outputs.
- avoid MATLAB figure or GUI code.

Deployed MATLAB code runs on the server, any figures or GUIs created during runtime will show up on the server machine, not the client machine. If figures or GUIs are required to run to create the function results, make sure to close these figures at the end of your code to avoid left over windows and leaking resources on the server.

### See Also

#### More About

- “State-Dependent Functions” on page 1-3
- “Write Deployable MATLAB Code” (MATLAB Compiler)



## State-Dependent Functions

MATLAB code that you want to deploy often carries state—a specific data value in a program or program variable.

### Does My MATLAB Function Carry State?

Example of carrying state in a MATLAB program include, but are not limited to:

- Modifying or relying on the MATLAB path and the Java® class path
- Accessing MATLAB state that is inherently persistent or global. Some example of this include:
  - Random number seeds
  - Handle Graphics® root objects that retain data
  - MATLAB or MATLAB toolbox settings and preferences
- Creating global and persistent variables.
- Loading MATLAB objects (MATLAB classes) into MATLAB. If you access a MATLAB object in any way, it loads into MATLAB.
- Calling MEX files, Java methods, or C# methods containing static variables.

### Defensive Coding Practices

If your MATLAB function not only carries state, but also *relies on it* for your function to properly execute, you must take additional steps (listed in this section) to ensure state retention.

When you deploy your application, consider cases where you carry state, and safeguard against that state's corruption if needed. *Assume* that your state may be changed and code defensively against that condition.

The following are examples of “defensive coding” practices:

#### Reset System-Generated Values in the Deployed Application

If you are using a random number seed, for example, reset it in your deployed application program to ensure the integrity of your original MATLAB function.

#### Validate Global or Persistent Variable Values

If you must use global or persistent variables, always validate their value in your deployed application and reset if needed.

#### Ensure Access to Data Caches

If your function relies on cached replies to previous requests, for instance, ensure your deployed system and application has access to that cache outside of the MATLAB environment.

#### Use Simple Data Types When Possible

Simple data types are usually not tied to a specific application and means of storing state. Your options for choosing an appropriate state-preserving tool increase as your data types become less complicated and specific.

### **Avoid Using MATLAB Callback Functions**

Avoid using MATLAB callbacks, such as `timer`. Callback functions have the ability to interrupt and override the current state of the MATLAB Production Server worker and may yield unpredictable results in multiuser environments.

### **Techniques for Preserving State**

The most appropriate method for preserving state depends largely on the type of data you need to save.

- Databases provide the most versatile and scalable means for retaining stateful data. The database acts as a generic repository and can generally work with any application in an enterprise development environment. It does not impose requirements or restrictions on the data structure or layout. Another related technique is to use comma-delimited files, in applications such as Microsoft® Excel®.
- Data that is specific to a third-party programming language, such as Java and C#, can be retained using a number of techniques. Consult the online documentation for the appropriate third-party vendor for best practices on preserving state.

---

**Caution** Using MATLAB `LOAD` and `SAVE` functions is often used to preserve state in MATLAB applications and workspaces. While this may be successful in some circumstances, it is highly recommended that the data be validated and reset if needed, if not stored in a generic repository such as a database.

---

## **Deploying MATLAB Functions Containing MEX Files**

If the MATLAB function you are deploying uses MEX files, ensure that the system running MATLAB Production Server is running the version of MATLAB Compiler used to create the MEX files.

Coordinate with your server administrator and application developer as needed.

## Supported MATLAB Data Types for Client and Server Marshaling

MATLAB Production Server supports and partially supports certain MATLAB data types for marshaling between client programs and server instances. However, certain MATLAB data types are unsupported.

### Supported Data Types

- Numeric types - Integers and floating-point numbers
- Character arrays
- Structures
- Cell arrays
- Logical

### Partially Supported Data Types

- Complex numbers — Only the Python® and C client libraries and the MATLAB Production Server “RESTful API” and JSON support complex numbers.
- String arrays, enumerations, and `datetime` arrays — Only the MATLAB Production Server RESTful API and JSON support these data types.

### Unsupported Data Types

Some of the MATLAB data types that MATLAB Production Server does not support include the following.

- MATLAB function handles
- Sparse matrices
- Tables
- Timetables

### See Also

#### More About

- “JSON Representation of MATLAB Data Types”

# Create a Deployable Archive from MATLAB Production Server Code

---

- “Create Deployable Archive for MATLAB Production Server” on page 2-2
- “Package Deployable Archives with Production Server Compiler App” on page 2-5
- “Package Deployable Archives from Command Line” on page 2-8
- “Modifying Deployed Functions” on page 2-10

# Create Deployable Archive for MATLAB Production Server

**Supported platform:** Windows®, Linux®, Mac

This example shows how to create a deployable archive from a MATLAB function. You can then give the generated archive to a system administrator to deploy it on the MATLAB Production Server environment.

## Create MATLAB Function

In MATLAB, examine the MATLAB program that you want to package.

For this example, write a function `addmatrix.m` as follows.

```
function a = addmatrix(a1, a2)

a = a1 + a2;
```

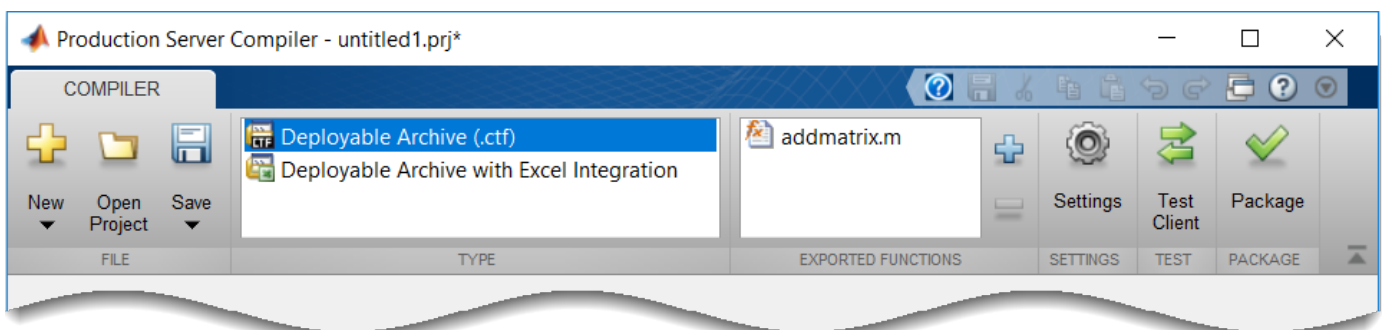
At the MATLAB command prompt, enter `addmatrix([1 4 7; 2 5 8; 3 6 9], [1 4 7; 2 5 8; 3 6 9])`.

The output is:


```
ans =
     2     8    14
     4    10    16
     6    12    18
```

## Create Deployable Archive with Production Server Compiler App

- 1 On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Production Server Compiler**. In the **Production Server Compiler** project window, click **Deployable Archive (.ctf)**.



Alternatively, you can open the **Production Server Compiler** app by entering `productionServerCompiler` at the MATLAB prompt.

- 2 In the **MATLAB Compiler SDK** project window, specify the main file of the MATLAB application that you want to deploy.
  - 1 In the **Exported Functions** section, click .
  - 2 In the **Add Files** window, browse to the example folder, and select the function you want to package.

Click **Open**.

The function `addmatrix.m` is added to the list of main files.

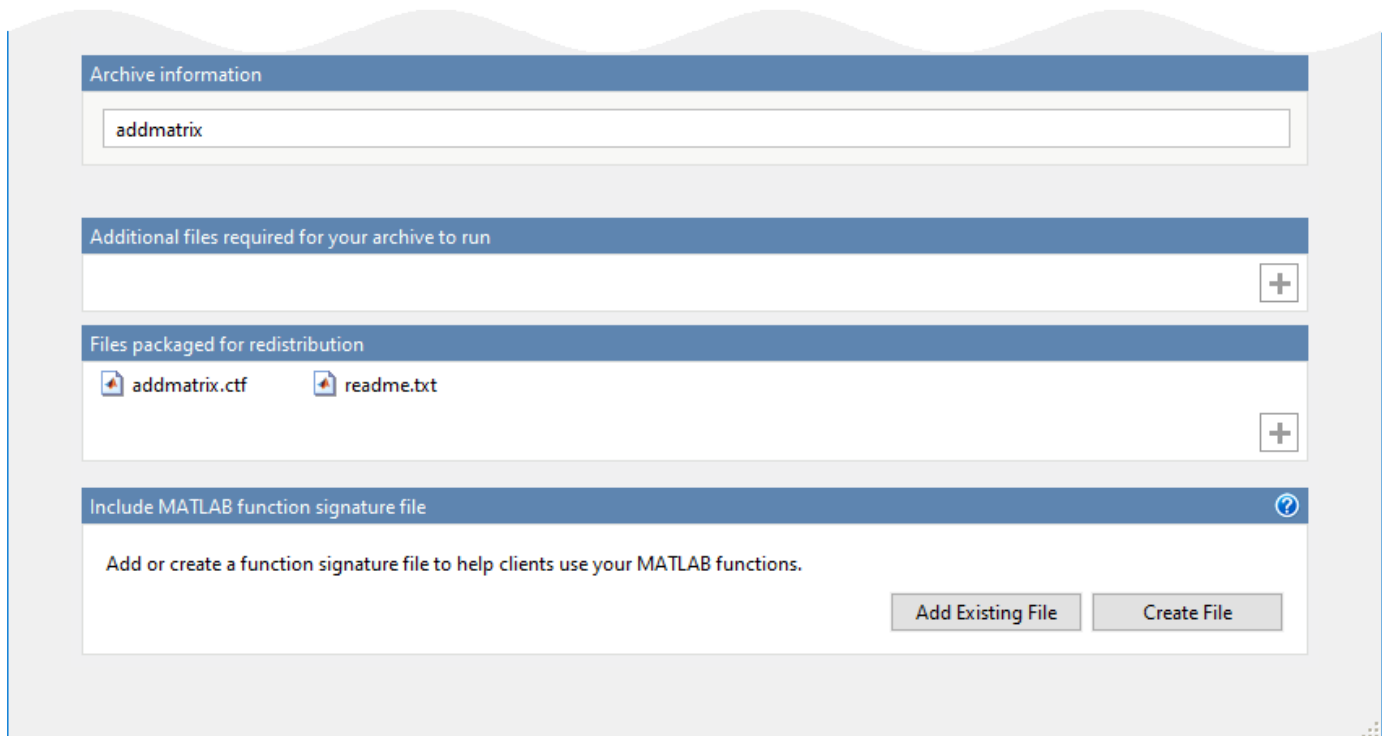
## Customize Application and Its Appearance

You can customize your deployable archive, and add more information about the application as follows:

- **Archive information** — Editable information about the deployed archive.
- **Additional files required for your archive to run** — Additional files required to run the generated archive. These files are included in the generated archive installer. See “Manage Required Files in Compiler Project” (MATLAB Compiler SDK).
- **Files packaged for redistribution** — Files that are installed with your archive. These files include:
  - Generated deployable archive
  - Generated `readme.txt`

See “Specify Files to Install with Application” (MATLAB Compiler SDK).

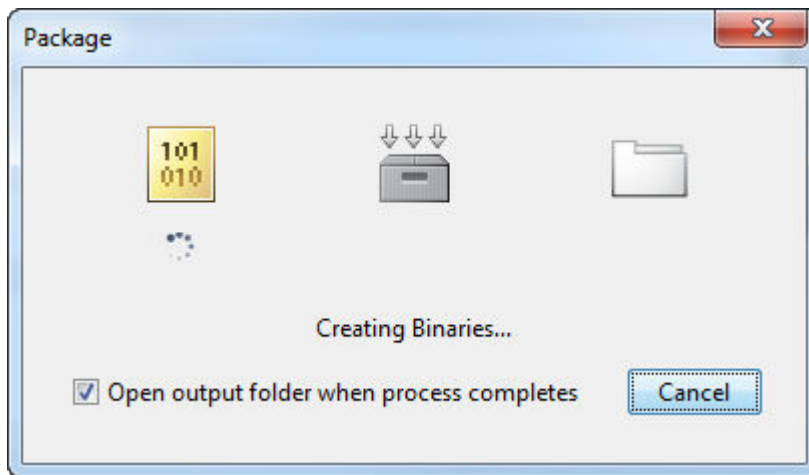
- **Include MATLAB function signature file** — Add or create a function signature file to help clients use your MATLAB functions. See “MATLAB Function Signatures in JSON”.



## Package Application

- 1 To generate the packaged application, click **Package**.

In the Save Project dialog box, specify the location to save the project.



- 2 In the **Package** dialog box, verify that **Open output folder when process completes** is selected.

When the deployment process is complete, examine the generated output.

- `for_redistribution` — Folder containing the archive `archiveName.ctf`
- `for_testing` — Folder containing the raw generated files to create the installer
- `PackagingLog.txt` — Log file generated by MATLAB Compiler

### See Also

`deploytool` | `mcc` | `productionServerCompiler`

### More About

- Production Server Compiler
- “Share Deployable Archive”
- “MATLAB Function Signatures in JSON”



# Package Deployable Archives with Production Server Compiler App

**Supported platform:** Windows, Linux, Mac

This example shows how to create a deployable archive from a MATLAB function. You can then hand the generated archive to a system administrator who will deploy it into MATLAB Production Server.

## Create Function In MATLAB

In MATLAB, examine the MATLAB program that you want packaged.

For this example, write a function `addmatrix.m` as follows.

```
function a = addmatrix(a1, a2)
a = a1 + a2;
```

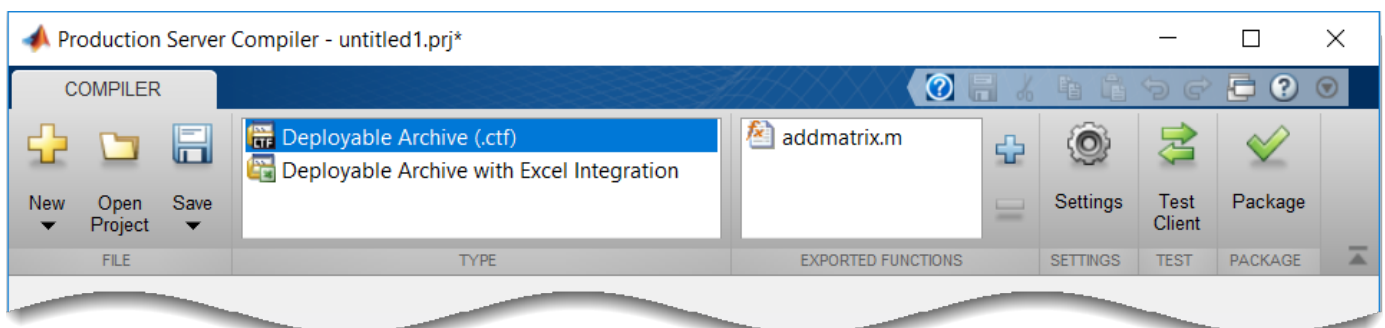
At the MATLAB command prompt, enter `addmatrix([1 4 7; 2 5 8; 3 6 9], [1 4 7; 2 5 8; 3 6 9])`.

The output is:


```
ans =
     2     8    14
     4    10    16
     6    12    18
```

## Create Deployable Archive with Production Server Compiler App

- 1 On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Production Server Compiler**. In the **Production Server Compiler** project window, click **Deployable Archive (.ctf)**.



Alternately, you can open the **Production Server Compiler** app by entering `productionServerCompiler` at the MATLAB prompt.

- 2 In the **MATLAB Compiler SDK** project window, specify the main file of the MATLAB application that you want to deploy.
  - 1 In the **Exported Functions** section of the toolbar, click .
  - 2 In the **Add Files** window, browse to the example folder, and select the function you want to package. Click **Open**.

The function `addmatrix.m` is added to the list of main files.

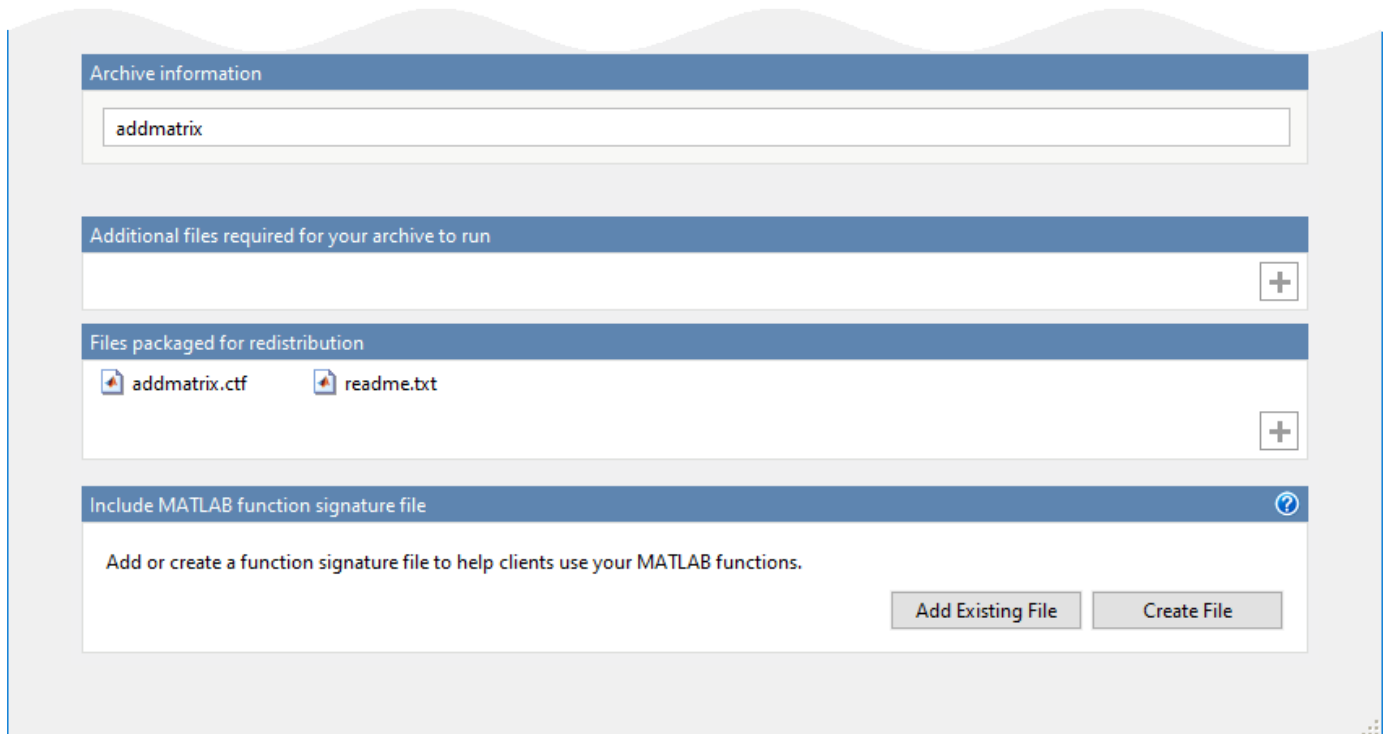
### Customize the Application and Its Appearance

You can customize your deployable archive, and add more information about the application as follows:

- **Archive information** — Editable information about the deployed archive.
- **Additional files required for your archive to run** — Additional files required by the generated archive to run. These files are included in the generated archive installer. See “Manage Required Files in Compiler Project” (MATLAB Compiler SDK).
- **Files packaged for redistribution** — Files that are installed with your application. These files include:
  - Generated deployable archive
  - Generated `readme.txt`

See “Specify Files to Install with Application” (MATLAB Compiler SDK)

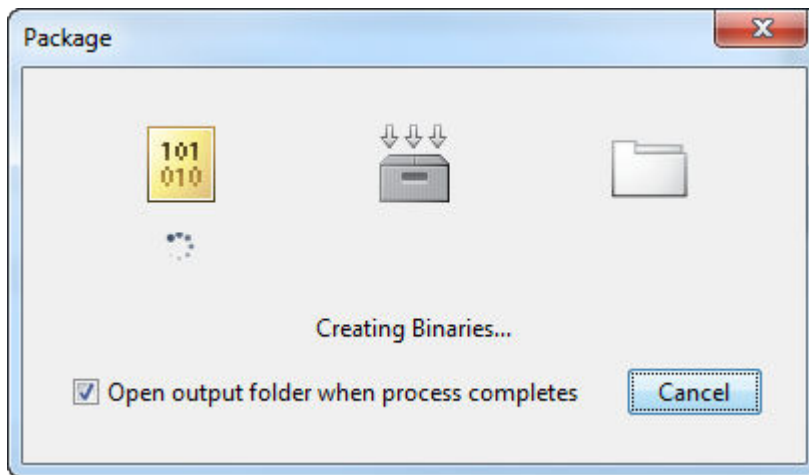
- **Include MATLAB function signature file** — Add or create a function signature file to help clients use your MATLAB functions.



### Package the Application

- 1 To generate the packaged application, click **Package**.

In the Save Project dialog box, specify the location to save the project.



- 2 In the **Package** dialog box, verify that the option **Open output folder when process completes** is selected.

When the deployment process is complete, examine the generated output.

- `for_redistribution` — A folder containing the installer to distribute the archive.
- `for_testing` — A folder containing the raw generated files to create the installer
- `PackagingLog.txt` — Log file generated by the packaging tool.

## See Also

`deploytool | mcc | productionServerCompiler`

## More About

- Production Server Compiler

## Package Deployable Archives from Command Line

### In this section...

“Execute Compiler Projects with `deploytool`” on page 2-8

“Package a Deployable Archive with `mcc`” on page 2-8

“Differences Between Compiler Apps and Command Line” on page 2-8

You can package deployable archives at the MATLAB prompt or your system prompt using either of these commands.

- `deploytool` invokes the Application Compiler app to execute a saved compiler project.
- `mcc` invokes the MATLAB Compiler to create a deployable application at the command prompt.

### Execute Compiler Projects with `deploytool`

The `deploytool` command has two flags that invoke one of the compiler apps to package an already existing project without opening a window.

- `-build project_name` — Invoke the correct compiler app to build the project but not generate an installer.
- `-package project_name` — Invoke the correct compiler app to build the project and generate an installer.

For example, `deploytool -package magicsquare` generates the binary files defined by the `magicsquare` project and packages them into an installer that you can distribute to others.

### Package a Deployable Archive with `mcc`

The `mcc` command invokes the MATLAB Compiler and provides fine-level control over the packaging of the deployable archive. It, however, cannot package the results in an installer.

To invoke the compiler to generate a deployable archive, use the `-W CTF:component_name` flag with `mcc`. The `-W CTF:component_name` flag creates a deployable archive called `component_name.ctf`.

For packaging deployable archives, you can also use the following options.

Option	Description
<code>-a filePath</code>	Add any files on the path to the generated binary.
<code>-d outFolder</code>	Specify the folder into which the results of packaging are written.
<code>class{className:mfilename...}</code>	Specify that an additional class is generated that includes methods for the listed MATLAB files.

### Differences Between Compiler Apps and Command Line

You perform the same functions using the compiler apps, a `compiler.build` function, or the `mcc` command-line interface. The interactive menus and dialog boxes used in the compiler apps build `mcc` commands that are customized to your specification. As such, your MATLAB code is processed the same way as if you were packaging it using `mcc`.

If you know the commands for the type of application you want to deploy and do not require an installer, it is faster to execute either `compiler.build` or `mcc` than go through the compiler app workflow.

Compiler app advantages include:

- You can perform related deployment tasks with a single intuitive interface.
- You can maintain related information in a convenient project file.
- Your project state persists between sessions.
- You can load previously stored compiler projects from a prepopulated menu.
- You can package applications for distribution.

### **See Also**

`deploytool | mcc`

### **More About**

- “Package Deployable Archives with Production Server Compiler App” on page 2-5

# Modifying Deployed Functions

After you have built a deployable archive, you are able to modify your MATLAB code, recompile, and see the change instantly reflected in the archive hosted on your server. This is known as hot deploying or redeploying a function.

To hot deploy, you must have a server created and running, with the built deployable archive located in the server's `auto_deploy` folder.

The server deploys the updated version of your archive when one of the following occurs:

- Compiled archive has an updated time stamp
- Change has occurred to the archive contents (new file or deleted file)

It takes a maximum of five seconds to redeploy a function using hot deployment. It takes a maximum of ten seconds to undeploy a function (remove the function from being hosted).

## See Also

`auto-deploy-root`

## More About

- “Share Deployable Archive”

# Customizing a Compiler Project

---

- “Customize an Application” on page 3-2
- “Manage Support Packages” on page 3-7

## Customize an Application

You can customize an application in several ways: customize the installer, manage files in the project, or add a custom installer path using the **Application Compiler** app or the **Library Compiler** app.

### Customize the Installer

#### Change Application Icon

To change the default icon, click the graphic to the left of the **Library name** or **Application name** field to preview the icon.



Click **Select icon**, and locate the graphic file to use as the application icon. Select the **Use mask** option to fill any blank spaces around the icon with white or the **Use border** option to add a border around the icon.

To return to the main window, click **Save and Use**.

#### Add Library or Application Information

You can provide further information about your application as follows:

- **Library/Application Name:** The name of the installed MATLAB artifacts. For example, if the name is `foo`, the installed executable is `foo.exe`, and the Windows start menu entry is **foo**. The folder created for the application is `InstallRoot/foo`.


The default value is the name of the first function listed in the **Main File(s)** field of the app.

- **Version:** The default value is 1.0.
- **Author name:** Name of the developer.
- **Support email address:** Email address to use for contact information.
- **Company name:** The full installation path for the installed MATLAB artifacts. For example, if the company name is `bar`, the full installation path would be `InstallRoot/bar/ApplicationName`.
- **Summary:** Brief summary describing the application.
- **Description:** Detailed explanation about the application.

All information is optional and, unless otherwise stated, is only displayed on the first page of the installer. On Windows systems, this information is also displayed in the Windows **Add/Remove Programs** control panel.



Library information





### Change the Splash Screen

The installer splash screen displays after the installer has started. It is displayed along with a status bar while the installer initializes.

You can change the default image by clicking the **Select custom splash screen**. When the file explorer opens, locate and select a new image.

You can drag and drop a custom image onto the default splash screen.

### Change the Installation Path

This table lists the default path the installer uses when installing the packaged binaries onto a target system.

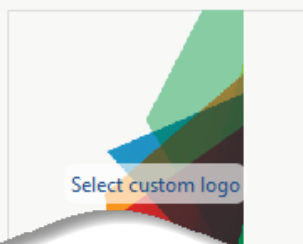
Windows	C:\Program Files\companyName\appName
Mac OS X	/Applications/companyName/appName
Linux	/usr/companyName/appName

You can change the default installation path by editing the **Default installation folder** field under **Additional installer options**.

Additional installer options

Default installation folder:

Installation notes



A text field specifying the path appended to the root folder is your installation folder. You can pick the root folder for the application installation folder. This table lists the optional custom root folders for each platform:

Windows	C:\Users\ <i>userName</i> \AppData
Linux	/usr/local

#### Change the Logo

The logo displays after the installer has started. It is displayed on the right side of the installer.

You change the default image in **Additional Installer Options** by clicking **Select custom logo**. When the file explorer opens, locate and select a new image. You can drag and drop a custom image onto the default logo.

#### Edit the Installation Notes

Installation notes are displayed once the installer has successfully installed the packaged files on the target system. You can provide useful information concerning any additional setup that is required to use the installed binaries and instructions for how to run the application.

### Manage Required Files in Compiler Project

The compiler uses a dependency analysis function to automatically determine what additional MATLAB files are required for the application to package and run. These files are automatically packaged into the generated binary. The compiler does not generate any wrapper code that allows direct access to the functions defined by the required files.

If you are using one of the compiler apps, the required files discovered by the dependency analysis function are listed in the **Files required for your application to run** or **Files required for your library to run** field.

To add files, click the plus button in the field, and select the file from the file explorer. To remove files, select the files, and press the **Delete** key.

---

**Caution** Removing files from the list of required files may cause your application to not package or not to run properly when deployed.

---

#### Using mcc

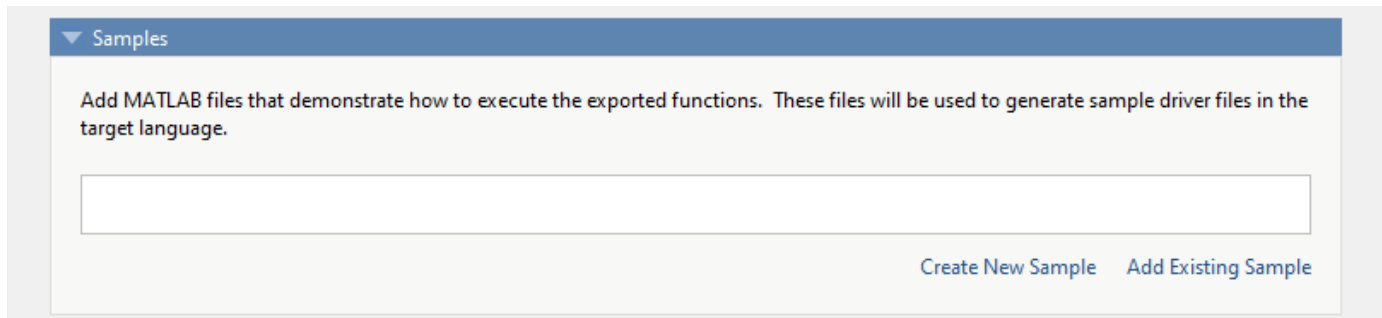
If you are using `mcc` to package your MATLAB code, the compiler does not display a list of required files before running. Instead, it packages all the required files that are discovered by the dependency analysis function and adds them to the generated binary file.

You can add files to the list by passing one or more `-a` arguments to `mcc`. The `-a` arguments add the specified files to the list of files to be added into the generated binary. For example, `-a hello.m` adds the file `hello.m` to the list of required files and `-a ./foo` adds all the files in `foo` and its subfolders to the list of required files.

## Sample Driver File Creation

The following target types support sample driver file creation in MATLAB Compiler SDK:

- C++ shared library
- Java package
- .NET assembly
- Python package



The sample driver file creation feature in **Library Compiler** uses MATLAB code to generate sample driver files in the target language. The sample driver files are used to implement the generated shared libraries into an application in the target language. In the app, click **Create New Sample** to automatically generate a new MATLAB script, or click **Add Existing Sample** to upload a MATLAB script that you have already written. After you package your functions, a sample driver file in the target language is generated from your MATLAB script and is saved in `for_redistribution_files_only\samples`. Sample driver files are also included in the installer in `for_redistribution`.

To automatically generate a new MATLAB file, click **Create New Sample**. This opens up a MATLAB file for you to edit. The sample file serves as a starting point, and you can edit it as necessary based on the behavior of your exported functions. The sample MATLAB files must follow these guidelines:

- The sample file code must use only exported functions.
- Each exported function must be in a separate sample file.
- Each call to the same exported function must be a separate sample file.
- The output of the exported function must be an n-dimensional numeric, char, logical, struct, or cell array.
- Data must be saved as a local variable and then passed to the exported function in the sample file code.
- Sample file code should not require user interaction.

Additional considerations specific to the target language are as follows:

- C++ `mwArray` API — `varargin` and `varargout` are not supported.
- .NET — Type-safe API is not supported.
- Python — Cell arrays and char arrays must be of size 1xN and struct arrays must be scalar. There are no restrictions on numeric or logical arrays, other than that they must be rectangular, as in MATLAB.

To upload a MATLAB file that you have already written, click **Add Existing Sample**. The MATLAB code should demonstrate how to execute the exported functions. The required MATLAB code can be only a few lines:

```
input1 = [1 4 7; 2 5 8; 3 6 9];  
input2 = [1 4 7; 2 5 8; 3 6 9];  
addoutput = addmatrix(input1,input2);
```


This code must also follow all the same guidelines outlined for the **Create New Sample** option.

You can also choose not to include a sample driver file at all during the packaging step. If you create your own driver code in the target language, you can later copy and paste it into the appropriate directory once the MATLAB functions are packaged.

## Specify Files to Install with Application

The compiler packages files to install along with the ones it generates. By default, the installer includes a readme file with instructions on installing the MATLAB Runtime and configuring it.

These files are listed in the **Files installed for your end user** section of the app.

To add files to the list, click , and select the file from the file explorer.

JAR files are added to the application class path as if you had called `javaaddpath`.

---

**Caution** Removing the binary targets from the list results in an installer that does not install the intended functionality.

---

When installed on a target computer, the files listed in the **Files installed for your end user** are saved in the application folder.

## Additional Runtime Settings

### See Also

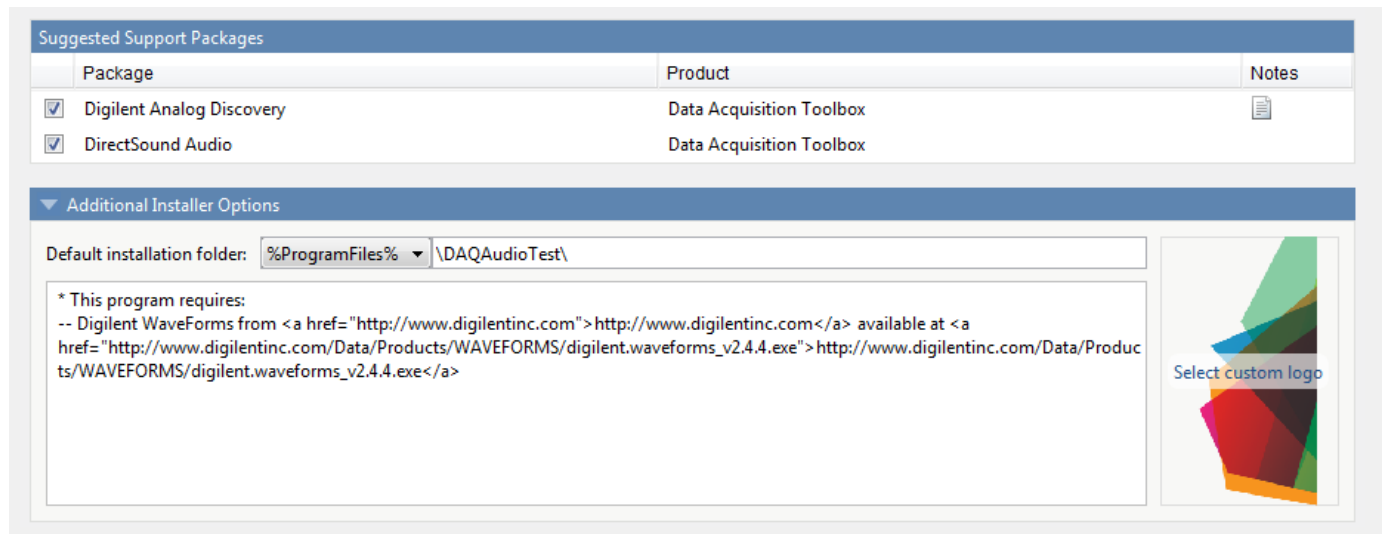
#### More About

- “Generate a C++ mxArray API Shared Library and Build a C++ Application” (MATLAB Compiler SDK)
- “Generate a C++ MATLAB Data API Shared Library and Build a C++ Application” (MATLAB Compiler SDK)

## Manage Support Packages

### Using a Compiler App

Many MATLAB toolboxes use support packages to interact with hardware or to provide additional processing capabilities. If your MATLAB code uses a toolbox with an installed support package, the app displays a **Suggested Support Packages** section.



The list displays all installed support packages that your MATLAB code requires. The list is determined using these criteria:

- the support package is installed
- your code has a direct dependency on the support package
- your code is dependent on the base product of the support package
- your code is dependent on at least one of the files listed as a dependency in the `mcc.xml` file of the support package, and the base product of the support package is MATLAB

Deselect support packages that are not required by your application.

Some support packages require third-party drivers that the compiler cannot package. In this case, the compiler adds the information to the installation notes. You can edit installation notes in the **Additional Installer Options** section of the app. To remove the installation note text, deselect the support package with the third-party dependency.

---

**Caution** Any text you enter beneath the generated text will be lost if you deselect the support package.

---

### Using the Command Line

Many MATLAB toolboxes use support packages to interact with hardware or to provide additional processing capabilities. If your MATLAB code uses a toolbox with an installed support package, use the `-a` flag with `mcc` command when packaging your MATLAB code to specify supporting files in the

support package folder. For example, if your function uses the OS Generic Video Interface support package, run the following command:

```
mcc -m -v test.m -a C:\MATLAB\SupportPackages\R2016b\toolbox\daq\supportpackages\daqaudio -a 'C:
```

Some support packages require third-party drivers that the compiler cannot package. In this case, you are responsible for downloading and installing the required drivers.

# Advanced Uses of the Command Line Compiler

---

- “Simplify Compilation Using Macros” on page 4-2
- “Invoke MATLAB Build Options” on page 4-4
- “MATLAB Runtime Component Cache and Deployable Archive Embedding” on page 4-6

## Simplify Compilation Using Macros

### In this section...

“Macros” on page 4-2

“Working With Macros” on page 4-2

### Macros

The compiler, through its exhaustive set of options, gives you access to the tools you need to do your job. If you want a simplified approach to compilation, you can use one simple *macro* that allows you to quickly accomplish basic compilation tasks. Macros let you group several options together to perform a particular type of compilation.

This table shows the relationship between the macro approach to accomplish a standard compilation and the multioption alternative.

Macro	Bundle	Creates	Option Equivalence Function Wrapper   Output Stage
-l	macro_option_l	Library	-W lib -T link:lib
-m	macro_option_m	Standalone application	-Wmain -Tlink:exe

### Working With Macros

The -m option tells the compiler to produce a standalone application. The -m macro is equivalent to the series of options

```
-W main -T link:exe
```

This table shows the options that compose the -m macro and the information that they provide to the compiler.

#### -m Macro

Option	Function
-W main	Produce a wrapper file suitable for a standalone application.
-T link:exe	Create an executable link as the output.

### Changing Macros

You can change the meaning of a macro by editing the corresponding `macro_option` file in `matlabroot\toolbox\compiler\bundles`. For example, to change the -m macro, edit the file `macro_option_m` in the `bundles` folder.

---

**Note** This changes the meaning of -m for all users of this MATLAB installation.

---



### Specifying Default Macros

As the MCCSTARTUP functionality has been replaced by bundle technology, the `macro_default` file that resides in `toolbox\compiler\bundles` can be used to specify default options to the compiler.

For example, adding `-mv` to the `macro_default` file causes the command:

```
mcc foo.m
```

to execute as though it were:

```
mcc -mv foo.m
```

Similarly, adding `-v` to the `macro_default` file causes the command:

```
mcc -W 'lib:libfoo' -T link:lib foo.m
```

to behave as though the command were:

```
mcc -v -W 'lib:libfoo' -T link:lib foo.m
```

## Invoke MATLAB Build Options

<b>In this section...</b>
---------------------------

“Specify Full Path Names to Build MATLAB Code” on page 4-4
--

“Using Bundles to Build MATLAB Code” on page 4-4
--

### Specify Full Path Names to Build MATLAB Code

If you specify a full path name to a MATLAB file on the `mcc` command line, the compiler

- 1 Breaks the full name into the corresponding path name and file names (`<path>` and `<file>`).
- 2 Replaces the full path name in the argument list with “`-I <path> <file>`”.

#### Specifying Full Path Names

For example:

```
mcc -m /home/user/myfile.m
```

would be treated as

```
mcc -m -I /home/user myfile.m
```

In rare situations, this behavior can lead to a potential source of confusion. For example, suppose you have two different MATLAB files that are both named `myfile.m` and they reside in `/home/user/dir1` and `/home/user/dir2`. The command

```
mcc -m -I /home/user/dir1 /home/user/dir2/myfile.m
```

would be equivalent to

```
mcc -m -I /home/user/dir1 -I /home/user/dir2 myfile.m
```

The compiler finds the `myfile.m` in `dir1` and compiles it instead of the one in `dir2` because of the behavior of the `-I` option. If you are concerned that this might be happening, you can specify the `-v` option and then see which MATLAB file the compiler parses. The `-v` option prints the full path name to the MATLAB file during the dependency analysis phase.

---

**Note** The compiler produces a warning (`specified_file_mismatch`) if a file with a full path name is included on the command line and the compiler finds it somewhere else.

---

### Using Bundles to Build MATLAB Code

Bundles provide a convenient way to group sets of compiler options and recall them as needed. The syntax of the bundle option is:

```
-B <bundle>[:<a1>,<a2>,...,<an>]
```

where `bundle` is either a predefined string such as `cpplib` or `csharedlib` or the name of a file that contains a set of `mcc` command-line options, arguments, filenames, and/or other `-B` options.

A bundle can include replacement parameters for compiler options that accept names and version numbers. For example, the bundle for C shared libraries, `csharedlib`, consists of:

```
-W lib:%1% -T link:lib
```

To invoke the compiler to produce the C shared library `mysharedlib` use:

```
mcc -B csharedlib:mysharedlib myfile.m myfile2.m
```

In general, each `%n%` in the bundle will be replaced with the corresponding option specified to the bundle. Use `%%` to include a `%` character. It is an error to pass too many or too few options to the bundle.

**Note** You can use the `-B` option with a replacement expression as is at the DOS or UNIX® prompt. If more than one parameter is passed, you must enclose the expression that follows the `-B` in single quotes. For example,

```
>>mcc -B csharedlib:libtimefun weekday data tic calendar toc
```

can be used as is at the MATLAB prompt because `libtimefun` is the only parameter being passed. If the example had two or more parameters, then the quotes would be necessary as in

```
>>mcc -B 'cexcel:component,class,1.0' ...
weekday data tic calendar toc
```

### Available Bundle Files

Bundle File	Creates	Contents
<code>cpplib</code>	C++ library	<code>-W cpplib:library_name -T link:lib</code>
<code>csharedlib</code>	C library	<code>-W lib:library_name -T link:lib</code>
<code>ccom</code>	COM component	<code>-W com:component_name,className,version -T link:lib</code>
<code>cexcel</code>	Excel Add-in	<code>-W excel:addin_name,className,version -T link:lib</code>
<code>cjava</code>	Java package	<code>-W java:packageName,className</code>
<code>dotnet</code>	.NET assembly	<code>-W dotnet:assembly_name,className,framework_version,security,remote_type -T link:lib</code>

## MATLAB Runtime Component Cache and Deployable Archive Embedding

### In this section...

“Overriding Default Behavior” on page 4-7

“For More Information” on page 4-7

Deployable archive data is automatically embedded directly in compiled components and extracted to a temporary folder.

Automatic embedding enables usage of MATLAB Runtime Component Cache features through environment variables.

These variables allow you to specify the following:

- Define the default location where you want the deployable archive to be automatically extracted
- Add diagnostic error printing options that can be used when automatically extracting the deployable archive, for troubleshooting purposes
- Tuning the MATLAB Runtime component cache size for performance reasons.

Use the following environment variables to change these settings.

Environment Variable	Purpose	Notes
MCR_CACHE_ROOT	When set to the location of where you want the deployable archive to be extracted, this variable overrides the default per-user component cache location. This is true for embedded <code>.ctf</code> files only.	On macOS, this variable is ignored in MATLAB R2020a and later. The app bundle contains the files necessary for runtime.
MCR_CACHE_SIZE	When set, this variable overrides the default component cache size.	The initial limit for this variable is 32M (megabytes). This may, however, be changed after you have set the variable the first time. Edit the file <code>.max_size</code> , which resides in the file designated by running the <code>mcrcachedir</code> command, with the desired cache size limit.

You can override this automatic embedding and extraction behavior by compiling with the “Overriding Default Behavior” on page 4-7 option.

**Caution** If you run `mcc` specifying conflicting wrapper and target types, the deployable archive will not be embedded into the generated component. For example, if you run:

```
mcc -W lib:myLib -T link:exe test.m test.c
```

the generated `test.exe` will not have the deployable archive embedded in it, as if you had specified a `-C` option to the command line.

## **Overriding Default Behavior**

To extract the deployable archive in a manner prior to R2008b, alongside the compiled .NET assembly, compile using the `mcc`'s `-C` option.

You might want to use this option to troubleshoot problems with the deployable archive, for example, as the log and diagnostic messages are much more visible.

## **For More Information**

For more information about the deployable archive, see “Deployable Archive” (MATLAB Compiler).



# Functions

---

## compiler.build.productionServerArchive

Create an archive for deployment to MATLAB Production Server

### Syntax

```
compiler.build.productionServerArchive(FunctionFiles)
compiler.build.productionServerArchive(FunctionFiles,Name,Value)
compiler.build.productionServerArchive(opts)
results = compiler.build.productionServerArchive( ___ )
```

### Description

`compiler.build.productionServerArchive(FunctionFiles)` creates a deployable archive using the MATLAB functions specified by `FunctionFiles`.

`compiler.build.productionServerArchive(FunctionFiles,Name,Value)` creates a deployable archive with additional options specified using one or more name-value arguments. Options include the archive name, JSON function signatures, and output directory.

`compiler.build.productionServerArchive(opts)` creates a deployable archive with options specified using a `compiler.build.ProductionServerArchiveOptions` object `opts`. You cannot specify any other options using name-value arguments.

`results = compiler.build.productionServerArchive( ___ )` returns build information as a `compiler.build.Results` object using any of the input argument combinations in previous syntaxes. The build information consists of the build type, the path to the compiled archive, and build options.

### Examples

#### Create Production Server Archive

Create a deployable server archive.

In MATLAB, locate the MATLAB function that you want to deploy as an archive. For this example, use the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
```

Build a production server archive using the `compiler.build.productionServerArchive` command.

```
compiler.build.productionServerArchive(appFile);
```

This syntax generates the following files within a folder named `mymagicproductionServerArchive` in your current working directory:

- `mymagic.ctf` — Deployable production server archive file.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see [MATLAB Compiler Limitations \(MATLAB Compiler\)](#).



- `readme.txt` — Readme file that contains information on deployment prerequisites and the list of files to package for deployment.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.

## Customize Production Server Archive

Create a production server archive and customize it using name-value arguments.

Build a production server archive using the `compiler.build.productionServerArchive` command. Use name-value arguments to specify the archive name and add a JSON signature file.

```
compiler.build.productionServerArchive(["myfunc1.m","myfunc2.m"],...
    'ArchiveName','MagicApp',...
    'FunctionSignatures','signatures.json');
```

## Create Multiple Production Server Archives Using Options Object

Customize multiple production server archives using a `compiler.build.ProductionServerArchiveOptions` object.

Create a `ProductionServerArchiveOptions` object using `example.m`. Use name-value arguments to specify a common output directory, disable automatically including data files, and enable verbose output.

```
opts = compiler.build.ProductionServerArchiveOptions('example.m',...
    'OutputDir','D:\Documents\MATLAB\work\ProductionServerBatch',...
    'AutoDetectDataFiles','off',...
    'Verbose','on');
```

```
opts =
```

ProductionServerArchiveOptions with properties:

```
    ArchiveName: 'example'
    FunctionFiles: {'D:\Documents\MATLAB\work\example.m'}
    FunctionSignatures: ''
    AdditionalFiles: {}
    AutoDetectDataFiles: off
    Verbose: on
    OutputDir: 'D:\Documents\MATLAB\work\ProductionServerBatch'
```

Build the production server archive using the `ProductionServerArchiveOptions` object.

```
compiler.build.productionServerArchive(opts);
```

To compile using the function file `example2.m` with the same options, use dot notation to modify the `FunctionFiles` of the existing `ProductionServerArchiveOptions` object before running the build function again.

```
opts.FunctionFiles = 'example2.m';
compiler.build.productionServerArchive(opts);
```

By modifying the `FunctionFiles` argument and recompiling, you can compile multiple archives using the same options object.

## Input Arguments

### **FunctionFiles** — Files implementing MATLAB functions

character vector | string scalar | cell array of character vectors | string array

Files implementing MATLAB functions, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a `.m` extension.

Example: `["myfunc1.m", "myfunc2.m"]`

Data Types: `char` | `string` | `cell`

### **opts** — Production server options object

`compiler.build.ProductionServerArchiveOptions` object

Production server archive build options, specified as a `compiler.build.ProductionServerArchiveOptions` object.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Verbose', 'on'`

### **ArchiveName** — Name of deployable archive

character vector | string scalar

Name of the deployable archive, specified as a character vector or a string scalar. The default name of the generated archive is the first entry of the `FunctionFiles` argument.

Example: `'ArchiveName', 'MyMagic'`

Data Types: `char` | `string`

### **AutoDetectDataFiles** — Flag to automatically include data files

`'on'` (default) | on/off logical value

Flag to automatically include data files, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then data files that you provide as inputs to certain functions (such as `load` and `fopen`) are automatically included in the production server archive.
- If you set this property to `'off'`, then you must add data files to the archive using the `AdditionalFiles` property.

Example: `'AutoDetectDataFiles', 'off'`

Data Types: `logical`

**FunctionSignatures — Path to JSON file**

character vector | string scalar

Path to a JSON file that details the signatures of all functions listed in `FunctionFiles`, specified as a character vector or a string scalar. For information on specifying function signatures, see “MATLAB Function Signatures in JSON”.

Example: 'FunctionSignatures', 'D:\Documents\MATLAB\work\magicapp\nsignatures.json'

Data Types: char | string

**OutputDir — Path to output directory**

character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the archive name appended with `productionServerArchive`.

Example: 'OutputDir', 'D:\Documents\MATLAB\work\MyMagicproductionServerArchive'

**Verbose — Build verbosity**

'off' (default) | on/off logical value

Build verbosity, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to 'off', then the command window does not display progress information.

Example: 'Verbose', 'off'

Data Types: logical

**Output Arguments****results — Build results**`compiler.build.Results` object

Build results, returned as a `compiler.build.Results` object. The `Results` object consists of:

- Build type, which is 'productionServerArchive'
- Path to the deployable archive file
- Build options, specified as a `ProductionServerArchiveOptions` object

**See Also**

`compiler.build.ProductionServerArchiveOptions` | `compiler.build.Results` | `productionServerCompiler`

**Introduced in R2020b**

# compiler.build.ProductionServerArchiveOptions

Options for building deployable archives

## Syntax

```
opts = compiler.build.ProductionServerArchiveOptions(FunctionFiles)
opts = compiler.build.ProductionServerArchiveOptions(FunctionFiles,
Name,Value)
```

## Description

`opts = compiler.build.ProductionServerArchiveOptions(FunctionFiles)` creates a `ProductionServerArchiveOptions` object using the MATLAB functions specified by `FunctionFiles`. Use the `ProductionServerArchiveOptions` object as an input to the `compiler.build.productionServerArchive` function.

`opts = compiler.build.ProductionServerArchiveOptions(FunctionFiles, Name,Value)` creates a `ProductionServerArchiveOptions` object with options specified using one or more name-value arguments. Options include the archive name, output directory, and additional files to include.

## Examples

### Create Deployable Archive Options Object

Create a `ProductionServerArchiveOptions` object from a function file.

For this example, use the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
opts = compiler.build.ProductionServerArchiveOptions(appFile)
```

```
opts =
```

`ProductionServerArchiveOptions` with properties:

```
    ArchiveName: 'magicsquare'
    FunctionFiles: {'C:\Program Files\MATLAB\R2021a\extern\examples\compiler\magicsquare.m'}
    FunctionSignatures: ''
    AdditionalFiles: {}
    AutoDetectDataFiles: on
    OutputDir: '.\magicsquareproductionServerArchive'
    Verbose: off
```

You can modify the property values of an existing `ProductionServerArchiveOptions` object using dot notation. For example, enable verbose output.

```
opts.Verbose = 'on'
```

```
opts =
```

ProductionServerArchiveOptions with properties:

```

    ArchiveName: 'magicsquare'
    FunctionFiles: {'C:\Program Files\MATLAB\R2021a\extern\examples\compiler\magicsquare.m'
FunctionSignatures: ''
    AdditionalFiles: {}
AutoDetectDataFiles: on
    OutputDir: '.\magicsquareproductionServerArchive'
    Verbose: on

```

Use the `DotNETAssemblyOptions` object as an input to the `compiler.build.productionServerArchive` function to build a production server archive.

```
compiler.build.productionServerArchive(opts);
```

### Customize Deployable Archive Options Object

Create a production server archive using a `ProductionServerArchiveOptions` object.

Create a `ProductionServerArchiveOptions` object using the function files `myfunc1.m` and `myfunc2.m`. Use name-value arguments to specify the output directory, enable verbose output, and disable automatic detection of data files.

```

opts = compiler.build.ProductionServerArchiveOptions(["myfunc1.m", "myfunc2.m"], ...
    'ArchiveName', 'MyServer', ...
    'OutputDir', 'D:\Documents\MATLAB\work\ProductionServer', ...
    'AutoDetectDataFiles', 'off')

```

```
opts =
```

ProductionServerArchiveOptions with properties:

```

    ArchiveName: 'MyServer'
    FunctionFiles: {2x1 cell}
FunctionSignatures: ''
    AdditionalFiles: {}
AutoDetectDataFiles: off
    OutputDir: 'D:\Documents\MATLAB\work\ProductionServer'
    Verbose: off

```

You can modify the property values of an existing `ProductionServerArchiveOptions` object using dot notation. For example, enable verbose output.

```
opts.Verbose = 'on'
```

```
opts =
```

ProductionServerArchiveOptions with properties:

```

    ArchiveName: 'MyServer'
    FunctionFiles: {2x1 cell}
FunctionSignatures: ''
    AdditionalFiles: {}
AutoDetectDataFiles: off
    OutputDir: 'D:\Documents\MATLAB\work\ProductionServer\'
    Verbose: on

```

Use the `ProductionServerArchiveOptions` object as an input to the `compiler.build.productionServerArchive` function to build a production server archive.

```
buildResults = compiler.build.productionServerArchive(opts);
```

## Input Arguments

### FunctionFiles — Files implementing MATLAB functions

character vector | string scalar | cell array of character vectors | string array

Files implementing MATLAB functions, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a `.m` extension.

Example: `["myfunc1.m", "myfunc2.m"]`

Data Types: `char` | `string` | `cell`

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Verbose', 'on'`

### ArchiveName — Name of deployable archive

character vector | string scalar

Name of the deployable archive, specified as a character vector or a string scalar. The default name of the generated archive is the first entry of the `FunctionFiles` argument.

Example: `'ArchiveName', 'MyMagic'`

Data Types: `char` | `string`

### AutoDetectDataFiles — Flag to automatically include data files

`'on'` (default) | on/off logical value

Flag to automatically include data files, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then data files that you provide as inputs to certain functions (such as `load` and `fopen`) are automatically included in the production server archive.
- If you set this property to `'off'`, then you must add data files to the archive using the `AdditionalFiles` property.

Example: `'AutoDetectDataFiles', 'off'`

Data Types: `logical`

### FunctionSignatures — Path to JSON file

character vector | string scalar

Path to a JSON file that details the signatures of all functions listed in `FunctionFiles`, specified as a character vector or a string scalar. For information on specifying function signatures, see “MATLAB Function Signatures in JSON”.

Example: 'FunctionSignatures', 'D:\Documents\MATLAB\work\magicapp\nsignatures.json'

Data Types: char | string

### **OutputDir — Path to output directory**

character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the archive name appended with `productionServerArchive`.

Example: 'OutputDir', 'D:\Documents\MATLAB\work\MyMagicproductionServerArchive'

### **Verbose — Build verbosity**

'off' (default) | on/off logical value

Build verbosity, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to `true`, and 'off' is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to 'off', then the command window does not display progress information.

Example: 'Verbose', 'off'

Data Types: logical

## **Output Arguments**

### **opts — Production server archive build options**

`ProductionServerArchiveOptions` object

Production server archive build options, returned as a `ProductionServerArchiveOptions` object.

## **See Also**

`compiler.build.productionServerArchive`

**Introduced in R2020b**



# compiler.build.Results

Compiler build results object

## Description

A `compiler.build.Results` object contains information about the build type, generated files, and build options of a `compiler.build` function.

All `Results` properties are read-only. You can use dot notation to query these properties.

## Creation

There are several ways to create a `compiler.build.Results` object.

## Properties

### BuildType – Build type

This property is read-only.

The build type of the `compiler.build` function used to generate the results, specified as a character vector:

<code>compiler.build</code> Function	Build Type
--------------------------------------	------------

Data Types: char

### Files – Paths to compiled files

cell array of character vectors

This property is read-only.

Paths to the compiled files of the `compiler.build` function used to generate the results, specified as a cell array of character vectors.

Build Type	Files
------------	-------

Example: {'D:\Documents\MATLAB\work\MagicSquareArchive\MagicSquare.ctf'}

Data Types: cell

### Options – Build options

This property is read-only.

Build options of the `compiler.build` function used to generate the results, specified as an options object of the corresponding build type.

<b>Build Type</b>	<b>Options</b>
-------------------	----------------

## **Examples**

## **See Also**

**Introduced in R2020b**

# productionServerCompiler

Test, build and package functions for use with MATLAB Production Server

## Syntax

```
productionServerCompiler  
productionServerCompiler project_name
```

## Description

`productionServerCompiler` opens the Production Server Compiler app for the creation of a new compiler project.

`productionServerCompiler project_name` opens the Production Server Compiler app with the project preloaded.

## Examples

### Create a New Production Server Project

Open the Production Server Compiler app to create a new project.

```
productionServerCompiler
```

## Input Arguments

**project\_name** — name of the project to be compiled

character array or string

Specify the name of a previously saved project. The project must be on the current path.

## Compatibility Considerations

**-build and -package options will be removed**

*Not recommended starting in R2020a*

The `-build` and `-package` options will be removed. To generate deployable archives, use the `compiler.build.productionServerArchive` function, or the `mcc` command, or the **Production Server Compiler** app.

**Introduced in R2014a**

## deploytool

Open a list of application deployment apps

### Syntax

```
deploytool  
deploytool project_name
```

### Description

`deploytool` opens a list of application deployment apps.

`deploytool project_name` opens the appropriate deployment app with the project preloaded.

### Examples

#### Open a List of Application Deployment Apps

Open the list of apps.

```
deploytool
```

### Input Arguments

#### **project\_name** — name of the project to be opened

character array or string

Name of the project to be opened by the appropriate deployment app, specified as a character array or string. The project must be on the current path.

### Compatibility Considerations

#### **-build and -package options will be removed**

*Not recommended starting in R2020a*

The `-build` and `-package` options will be removed. To build applications, use one of the `compiler.build` family of functions or the `mcc` command; and to package and create an installer, use the `compiler.package.installer` function.

## mcc

Compile MATLAB functions for deployment

### Syntax

```
mcc options mfilename1 mfilename2...mfilenameN
```

```
mcc -W CTF:archive_name -U options mfilename1 mfilename2...mfilenameN
```

```
mcc -W mpsxl:addin_name,className,version input_marshallng_flags
output_marshallng_flags -T link:lib options mfilename1
mfilename2...mfilenameN
```

### Description

#### General Usage

`mcc options mfilename1 mfilename2...mfilenameN` compiles the functions as specified by the options. The options used depend on the intended results of the compilation.

For information on compiling:

- standalone applications, Excel add-ins, or Hadoop® jobs, see `mcc` for MATLAB Compiler
- C/C++ shared libraries, .NET assemblies, Java packages, or Python packages, see `mcc` for MATLAB Compiler SDK

#### Deployable Archive for MATLAB Production Server

`mcc -W CTF:archive_name -U options mfilename1 mfilename2...mfilenameN` instructs the compiler to create a deployable archive (.ctf file) for use with a MATLAB Production Server instance.

The syntax also creates the server-side deployable archive (.ctf file) for Microsoft Excel add-ins.

#### Excel Add-In for MATLAB Production Server

`mcc -W mpsxl:addin_name,className,version input_marshallng_flags output_marshallng_flags -T link:lib options mfilename1 mfilename2...mfilenameN` creates a client-side Microsoft Excel add-in from the specified files that can be used to send requests to MATLAB Production Server from Excel. Creating the client-side add-in *must* be preceded by creating a server-side deployable archive (.ctf file) from the specified files. A purely client side add-in is not viable.

- *addin\_name* — Specifies the name of the add-in.
- *className* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the *addin\_name* as the default.
- *version* — Specifies the version of the add-in specified as *major.minor*.
  - *major* — Specifies the major version number. If you do not specify a version number, `mcc` uses the latest version.

- *minor* — Specifies the minor version number. If you do not specify a version number, `mcc` uses the latest version.
- *input\_marshaling\_flags* — Specifies options for how data is marshaled between Microsoft Excel and MATLAB.
  - `-replaceBlankWithNaN` — Specifies that a blank in Microsoft Excel is marshaled into NaN in MATLAB. If you do not specify this flag, blanks are marshaled into 0.
  - `-convertDateToString` — Specifies that dates in Microsoft Excel are marshaled into MATLAB character vectors. If you do not specify this flag, dates are marshaled into MATLAB doubles.
- *output\_marshaling\_flags* — Specifies options for how data is marshaled between MATLAB and Microsoft Excel.
  - `-replaceNaNWithZero` — Specifies that NaN in MATLAB is marshaled into a 0 in Microsoft Excel. If you do not specify this flag, NaN is marshaled into #QNAN in Visual Basic®.
  - `-convertNumericToDate` — Specifies that MATLAB numeric values are marshaled into Microsoft Excel dates. If you do not specify this flag, Microsoft Excel does not receive dates as output.

## Examples

### Create an Excel add-in for MATLAB Production Server

```
mcc -W 'mpsxl:myDeployableArchvie,myExcelClass,version=1.0' -T link:lib mymagic.m
```

## Input Arguments

### `mfilename1 mfilename2...mfilenameN` — Files to be compiled

list of file names

One or more files to be compiled, specified as a space-separated list of file names.

### options — Options for customizing the output

```
-a | -b | -B | -c | -C | -d | -f | -g | -G | -I | -K | -m | -M | -n | -N | -o | -p | -r | -R | -S | -T | -u | -U | -v | -w | -W | -X | -Y
```

Options for customizing the output, specified as a list of character vectors or string scalars.

- **-a**

Add files to the deployable archive using `-a path` to specify the files to be added. Multiple `-a` options are permitted.

If a file name is specified with `-a`, the compiler looks for these files on the MATLAB path, so specifying the full path name is optional. These files are not passed to `mbuild`, so you can include files such as data files.

If a folder name is specified with the `-a` option, the entire contents of that folder are added recursively to the deployable archive. For example,

```
mcc -m hello.m -a ./testdir
```

specifies that all files in `testdir`, as well as all files in its subfolders, are added to the deployable archive. The folder subtree in `testdir` is preserved in the deployable archive.

If the filename includes a wildcard pattern, only the files in the folder that match the pattern are added to the deployable archive and subfolders of the given path are not processed recursively. For example,

```
mcc -m hello.m -a ./testdir/*
```

specifies that all files in `./testdir` are added to the deployable archive and subfolders under `./testdir` are not processed recursively.

```
mcc -m hello.m -a ./testdir/*.m
```

specifies that all files with the extension `.m` under `./testdir` are added to the deployable archive and subfolders of `./testdir` are not processed recursively.

---

**Note** `*` is the only supported wildcard.

---

When you add files to the archive using `-a` that do not appear on the MATLAB path at the time of compilation, a path entry is added to the application's run-time path so that they appear on the path when the deployed code executes.

When you use the `-a` option to specify a full path to a resource, the basic path is preserved, with some modifications, but relative to a subdirectory of the runtime cache directory, not to the user's local folder. The cache directory is created from the deployable archive the first time the application is executed. You can use the function to determine whether the application is being run in deployed mode, and adjust the path accordingly. The `-a` option also creates a `.auth` file for authorization purposes.

---

**Caution** If you use the `-a` flag to include a file that is not on the MATLAB path, the folder containing the file is added to the MATLAB dependency analysis path. As a result, other files from that folder might be included in the compiled application.

---



---

**Note** If you use the `-a` flag to include custom Java classes, standalone applications work without any need to change the `classpath` as long as the Java class is not a member of a package. The same applies for JAR files. However, if the class being added is a member of a package, the MATLAB code needs to make an appropriate call to `javaaddpath` to update the `classpath` with the parent folder of the package.

---

- **-b**

Generate a Visual Basic file (`.bas`) containing the Microsoft Excel Formula Function interface to the COM object generated by MATLAB Compiler. When imported into the workbook Visual Basic code, this code allows the MATLAB function to be seen as a cell formula function.

- **-B**

Replace the file on the `mcc` command line with the contents of the specified file. Use

```
-B filename[:<a1>,<a2>,...,<an>]
```

The bundle `filename` should contain only `mcc` command-line options and corresponding arguments and/or other file names. The file might contain other `-B` options. A bundle can include

replacement parameters for compiler options that accept names and version numbers. See “Using Bundles to Build MATLAB Code” (MATLAB Compiler SDK).

- **-c**

When used in conjunction with the `-l` option, suppresses compiling and linking of the generated C wrapper code. The `-c` option cannot be used independently of the `-l` option.

- **-C**

Do not embed the deployable archive in binaries.

---

**Note** The `-C` flag is ignored for Java libraries.

---

- **-d**

Place output in a specified folder. Use

`-d outFolder`

to direct the generated files to *outFolder*. The specified folder must already exist.

- **-e**

Use `-e` in place of the `-m` option to generate a standalone Windows application that does not open a Windows command prompt on execution. `-e` is equivalent to `-W WinMain -T link:exe`.

This option works only on Windows operating systems.

- **-f**

Override the default options file with the specified options file. It specifically applies to the C/C++ shared libraries, COM, and Excel targets. Use

`-f filename`

to specify *filename* as the options file when calling `mbuild`. This option lets you use different ANSI compilers for different invocations of the compiler. This option is a direct pass-through to `mbuild`.

- **-g, -G**

Include debugging symbol information for the C/C++ code generated by MATLAB Compiler SDK. It also causes `mbuild` to pass appropriate debugging flags to the system C/C++ compiler. The debug option lets you backtrace up to the point where you can identify if the failure occurred in the initialization of MATLAB Runtime, the function call, or the termination routine. This option does not let you debug your MATLAB files with a C/C++ debugger.

- **-I**

Add a new folder path to the list of included folders. Each `-I` option appends the folder to the end of the list of paths to search. For example,

`-I <directory1> -I <directory2>`

sets up the search path so that *directory1* is searched first for MATLAB files, followed by *directory2*. This option is important for standalone compilation where the MATLAB path is not available.



If used in conjunction with the `-N` option, the `-I` option adds the folder to the compilation path in the same position where it appeared in the MATLAB path rather than at the head of the path.

- **-K**

Direct `mcc` to not delete output files if the compilation ends prematurely due to error.

The default behavior of `mcc` is to dispose of any partial output if the command fails to execute successfully.

- **-m**

Direct `mcc` to generate a standalone application.

- **-M**

Define compile-time options. Use

`-M string`

to pass `string` directly to `mbuild`. This option provides a useful mechanism for defining compile-time options, for example, `-M "-Dmacro=value"`.

---

**Note** Multiple `-M` options do not accumulate; only the rightmost `-M` option is used.

---

To pass options such as `/bigobj`, delineate the string according to your platform.

Platform	Syntax
MATLAB	<code>-M 'COMPFLAGS=\$COMPFLAGS /bigobj'</code>
Windows command prompt	<code>-M COMPFLAGS="\$COMPFLAGS /bigobj"</code>
Linux and macOS command line	<code>-M CFLAGS='\$CFLAGS /bigobj'</code>

- **-n**

The `-n` option automatically identifies numeric command line inputs and treats them as MATLAB doubles.

- **-N**

Passing `-N` clears the path of all folders except the following core folders (this list is subject to change over time):

- `matlabroot\toolbox\matlab`
- `matlabroot\toolbox\local`
- `matlabroot\toolbox\compiler`
- `matlabroot\toolbox\shared\bigdata`

Passing `-N` also retains all subfolders in this list that appear on the MATLAB path at compile time. Including `-N` on the command line lets you replace folders from the original path, while retaining the relative ordering of the included folders. All subfolders of the included folders that appear on the original path are also included. In addition, the `-N` option retains all folders that you included on the path that are not under `matlabroot\toolbox`.

When using the `-N` option, use the `-I` option to force inclusion of a folder, which is placed at the head of the compilation path. Use the `-p` option to conditionally include folders and their

subfolders; if they are present in the MATLAB path, they appear in the compilation path in the same order.

- **-o**

Specify the name of the final executable (standalone applications only). Use

`-o outputfile`

to name the final executable output of MATLAB Compiler. A suitable platform-dependent extension is added to the specified name (for example, `.exe` for Windows standalone applications).

- **-p**

Use in conjunction with the option `-N` to add specific folders and subfolders under `matlabroot\toolbox` to the compilation MATLAB path. The files are added in the same order in which they appear in the MATLAB path. Use the syntax

`-N -p directory`

where `directory` is the folder to be included. If `directory` is not an absolute path, it is assumed to be under the current working folder.

- If a folder is included with `-p` that is on the original MATLAB path, the folder and all its subfolders that appear on the original path are added to the compilation path in the same order.
- If a folder is included with `-p` that is not on the original MATLAB path, that folder is ignored. (You can use `-I` to force its inclusion.)

- **-r**

Embed resource icon in binary. The syntax is as follows:

`-r 'path/to/my_icon.ico'`

- **-R**

Provide MATLAB Runtime options.

---

**Note** This option is relevant only when building standalone applications using MATLAB Compiler.

---

The syntax is as follows:

`-R option`

Option	Description	Target
<code>' -logfile, filename'</code>	Specify a log file name. Option must be in single quotes.	MATLAB Compiler
<code>-nodisplay</code>	Suppress the MATLAB <code>nodisplay</code> run-time warning.	MATLAB Compiler

Option	Description	Target
-nojvm	Do not use the Java Virtual Machine (JVM).	MATLAB Compiler
-startmsg	Customizable user message displayed at initialization time.	MATLAB Compiler Standalone Applications
-completemsg	Customizable user message displayed when initialization is complete.	MATLAB Compiler Standalone Applications
-singleCompThread	Limit MATLAB to a single computational thread.	MATLAB Compiler

**Caution** When running on macOS, if you use `-nodisplay` as one of the options included in `mclInitializeApplication`, then the call to `mclInitializeApplication` must occur before calling `mclRunMain`.

**Note** If you specify the `-R` option for libraries created from MATLAB Compiler SDK, `mcc` still compiles without errors and generates the results. But the `-R` option doesn't apply to these libraries and does not do anything.

- **-S**

The standard behavior for the MATLAB Runtime is that every instance of a class gets its own MATLAB Runtime context. The context includes a global MATLAB workspace for variables, such as the path and a base workspace for each function in the class. If multiple instances of a class are created, each instance gets an independent context. This ensures that changes made to the global or base workspace in one instance of the class does not affect other instances of the same class.

In a singleton MATLAB Runtime, all instances of a class share the context. If multiple instances of a class are created, they use the context created by the first instance which saves startup time and some resources. However, any changes made to the global workspace or the base workspace by one instance impacts all class instances. For example, if `instance1` creates a global variable `A` in a singleton MATLAB Runtime, then `instance2` can use variable `A`.

Singleton MATLAB Runtime is only supported by the following products on these specific targets:

Target supported by Singleton MATLAB Runtime	Create a Singleton MATLAB Runtime by....
Excel add-in	Default behavior for target is singleton MATLAB Runtime. You do not need to perform other steps.
.NET assembly	Default behavior for target is singleton MATLAB Runtime. You do not need to perform other steps.
COM component	<ul style="list-style-type: none"> <li>• Using the Library Compiler app, click <b>Settings</b> and add <code>-S</code> to the <b>Additional parameters passed to MCC</b> field.</li> <li>• Using <code>mcc</code>, pass the <code>-S</code> flag.</li> </ul>
Java package	

- **-T**

Specify the output target phase and type.

Use the syntax `-T target` to define the output type.

Target	Description
compile:exe	Generate a C/C++ wrapper file, and compile C/C++ files to an object form suitable for linking into a standalone application.
compile:lib	Generate a C/C++ wrapper file, and compile C/C++ files to an object form suitable for linking into a shared library or DLL.
link:exe	Same as compile:exe and also link object files into a standalone application.
link:lib	Same as compile:lib and also link object files into a shared library or DLL.

- **-u**

Register COM component for the current user only on the development machine. The argument applies only to the generic COM component and Microsoft Excel add-in targets.

- **-U**

Build deployable archive (.ctf file) for MATLAB Production Server.

- **-v**

Display the compilation steps, including:

- MATLAB Compiler version number
- The source file names as they are processed
- The names of the generated output files as they are created
- The invocation of mbuild

The -v option passes the -v option to mbuild and displays information about mbuild.

- **-w**

Display warning messages. Use the syntax

`-w option [:<msg>]`

to control the display of warnings.

Syntax	Description
<code>-w list</code>	List the compile-time warnings that have abbreviated identifiers, together with their status.
<code>-w enable</code>	Enable all compile-time warnings.
<code>-w disable[:&lt;string&gt;]</code>	Disable specific compile-time warnings associated with <string>. Omit the optional <string> to apply the disable action to all compile-time warnings.
<code>-w enable[:&lt;string&gt;]</code>	Enable specific compile-time warnings associated with <string>. Omit the optional <string> to apply the enable action to all compile-time warnings.

Syntax	Description
<code>-w error[:&lt;string&gt;]</code>	Treat specific compile-time and runtime warnings associated with <i>&lt;string&gt;</i> as an error. Omit the optional <i>&lt;string&gt;</i> to apply the <code>error</code> action to all compile-time and runtime warnings.
<code>-w off[:&lt;string&gt;]</code>	Turn off warnings for specific error messages defined by <i>&lt;string&gt;</i> . Omit the optional <i>&lt;string&gt;</i> to apply the <code>off</code> action to all runtime warnings.
<code>-w on[:&lt;string&gt;]</code>	Turn on runtime warnings associated with <i>&lt;string&gt;</i> . Omit the optional <i>&lt;string&gt;</i> to apply the <code>on</code> action to all runtime warnings. This option is enabled by default.

You can also turn warnings on or off in your MATLAB code.

For example, to turn off warnings for deployed applications (specified using `isdeployed`) in `startup.m`, you write:

```
if isdeployed
    warning off
end
```

To turn on warnings for deployed applications, you write:

```
if isdeployed
    warning on
end
```

You can also specify multiple `-w` options.

For example, if you want to disable all warnings except `repeated_file`, you write:

```
-w disable -w enable:repeated_file
```

When you specify multiple `-w` options, they are processed from left to right.

- **-W**

Control the generation of function wrappers. Use the syntax

```
-W type
```

to control the generation of function wrappers for a collection of MATLAB files generated by the compiler. You provide a list of functions, and the compiler generates the wrapper functions and any appropriate global variable definitions.

Target	Syntax
Standalone Application	<code>-W 'main:appName,version=version'</code>
Standalone Application (no Windows console)	<code>-W 'WinMain:appName,version=version'</code>
Excel Add-In	<code>-W 'excel:addinName,className,version=version'</code>

Target	Syntax
C Shared Library	-W 'lib:libName'
C++ Shared Library	-W 'cpplib:libName[, {all legacy generic}]'
COM Component	-W 'com:comName,className'
.NET Assembly	-W 'dotnet:assemblyName,className,frameworkVersion,security,{remote local}'
Java Package	-W 'java:packageName,className'
Python Package	-W 'python:packageName,className'
MATLAB Production Server Deployable Archive	-W 'CTF:archiveName'
MATLAB Production Server Excel Add-In	-W 'mpsx1:addinName,className,version'

---

**Note** Replace single quotes with double when executing the command from a Windows Command Prompt.

---

- -X

Use -X to ignore data files read by common MATLAB file I/O functions during dependency analysis. For more information, see “Dependency Analysis Using MATLAB Compiler” (MATLAB Compiler). For examples on how to use the -X option, see `%#exclude`.

- -Y

Use

```
-Y license.lic
```

to override the default license file with the specified argument.

---

**Note** The -Y flag works only with the command-line mode.

---

```
>>!mcc -m foo.m -Y license.lic
```

---

## Tips

## See Also

# Apps

---

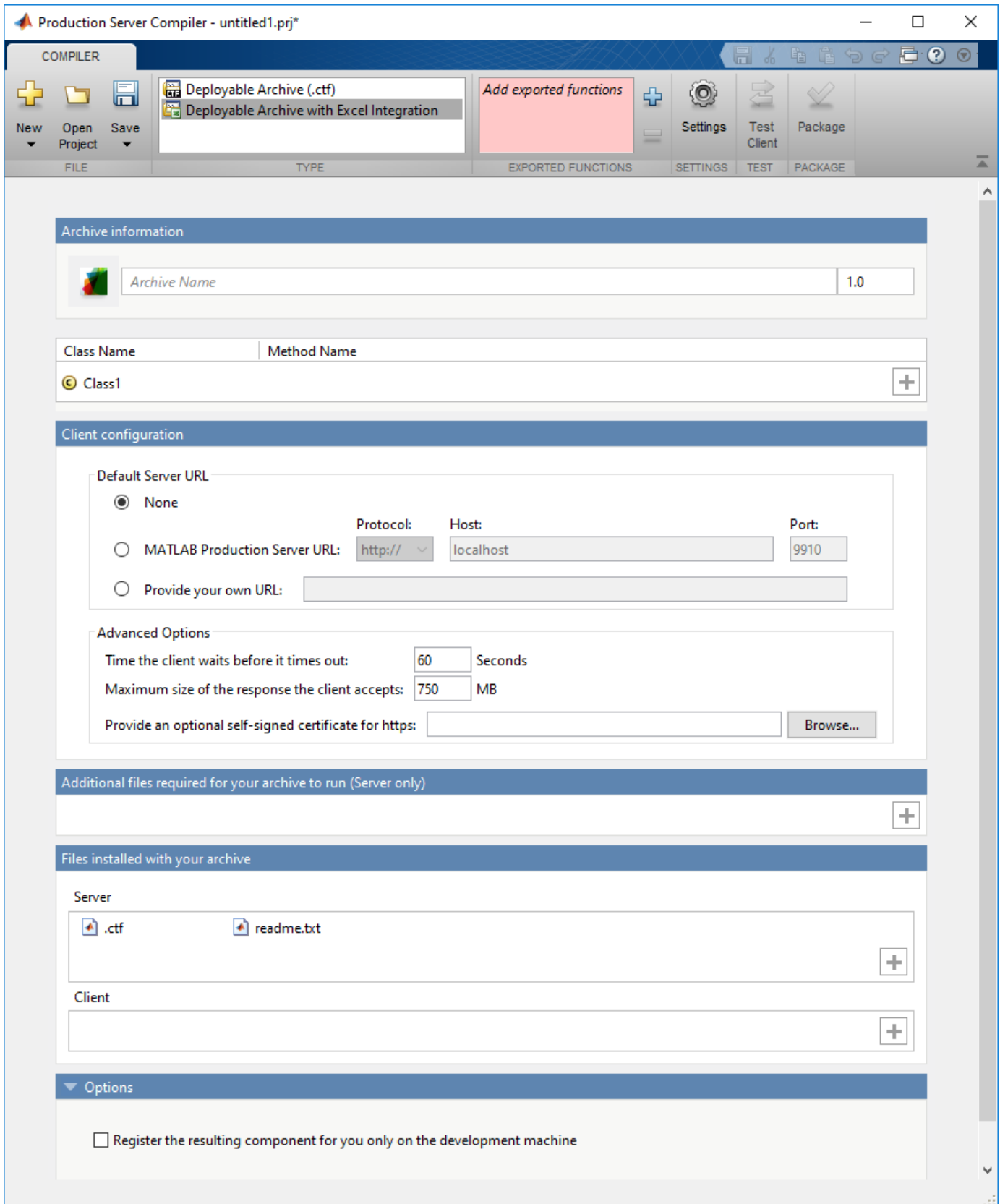
## Production Server Compiler

Package MATLAB programs for deployment to MATLAB Production Server

### Description

The **Production Server Compiler** app tests the integration of client code with MATLAB functions. It also packages MATLAB functions into archives for deployment to MATLAB Production Server.





## Open the Production Server Compiler App

- MATLAB Toolstrip: On the **Apps** tab, under **Application Deployment**, click the app icon.
- MATLAB command prompt: Enter `productionServerCompiler`.

## Examples

- “Create Deployable Archive for MATLAB Production Server” on page 2-2
- “Create and Install a Deployable Archive with Excel Integration For MATLAB Production Server”

## Parameters

### **type — type of archive generated**

Deployable Archive | Deployable Archive with Excel Integration

Type of archive to generate as a character array.

### **exported functions — functions to package**

list of character arrays

Functions to package as a list of character arrays.

### **archive information — name of the archive**

character array

Name of the archive as a character array.

### **files required for your archive to run — files that must be included with archive**

list of files

Files that must be included with archive as a list of files.

### **files packaged with the archive — optional files installed with archive**

list of files

Optional files installed with archive as a list of files.

### **Settings**

#### **Additional parameters passed to MCC — flags controlling the behavior of the compiler**

character array

Flags controlling the behavior of the compiler as a character array.

#### **testing files — folder where files for testing are stored**

character array

Folder where files for testing are stored as a character array.

#### **end user files — folder where files for building a custom installer are stored**

character array

Folder where files for building a custom installer are stored are stored as a character array.

**packaged installers — folder where generated installers are stored**

character array

Folder where generated installers are stored as a character array.

## **Programmatic Use**

productionServerCompiler

## **See Also**

### **Topics**

“Create Deployable Archive for MATLAB Production Server” on page 2-2

“Create and Install a Deployable Archive with Excel Integration For MATLAB Production Server”

**Introduced in R2013b**



# Persistence

---

## Use a Data Cache to Persist Data

Persistence provides a mechanism to cache data between calls to MATLAB code running on a server instance. A *persistence service* runs separately from the server instance and can be started and stopped manually. A *connection name* links a server instance to a persistence service. A persistence service uses a *persistence provider* to store data. Currently, Redis™ is the only supported persistence provider. The connection name is used in MATLAB application code to create a *data cache* in the linked persistence service.

Before starting a persistence service for an on-premises server instance from the system command prompt, you must create a JSON file called `mps_cache_config` and place it in the `config` folder of the server instance.

### `mps_cache_config`

```
{
  "Connections": {
    "<connection_name>": {
      "Provider": "Redis",
      "Host": "<hostname>",
      "Port": <port_number>,
      "Key": <access_key>
    }
  }
}
```

Specify the `<connection_name>`, `<hostname>`, and `<port_number>` in the JSON file. The host name can either be `localhost` or a remote host name obtained from an Azure® Redis cache resource. If you use Azure Cache for Redis, you must specify an access key. To use an Azure Redis cache, you need a Microsoft Azure account.

You can specify multiple connections in the file `mps_cache_config`. Each connection must have a unique name and a unique (host, port) pair. If you are using the persistence service through the dashboard, the file `mps_cache_config` is automatically created in the `config` folder of the server instance.

## Workflow to Use Persistence

Steps	Command Line	Dashboard
1. Create file <code>mps_cache_config</code>	Manually create a JSON file and place it in the <code>config</code> folder of the server instance.	Automatically created.
2. Start persistence service	Use <code>mps - cache</code> to start a persistence service.  For testing purposes, you can create a persistence service controller object using <code>mps . cache . control</code> .	<ul style="list-style-type: none"> <li>• Create a persistence service.</li> <li>• Add the persistence service to a server instance using a connection name.</li> <li>• Start the persistence service.</li> <li>• Attach the connection associated with a persistence service to a server instance.</li> </ul>
3. Create a data cache	Use <code>mps . cache . connect</code> to create a data cache.	Use <code>mps . cache . connect</code> to create a data cache.

## Example: Increment a Counter Using a Data Cache

This example shows you how to use persistence to increment a counter using a data cache. The example presents two workflows: a testing workflow that uses the MATLAB and a deployment workflow that requires an active server instance.

### Testing Workflow

- 1 Create a persistence service that uses Redis as the persistence provider and start the service.

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519)
start(ctrl)
```

- 2 Write MATLAB code that creates a cache and then updates a counter using the cache. Name the file `myCounter.m`

#### **myCounter.m**

```
function x = myCounter(cacheName,connectionName)

% create a data cache
c = mps.cache.connect(cacheName,'Connection',connectionName);

% if the key 'count' doesn't exist yet, initialize it
if isKey(c,'count') == false
    put(c,'count',0)
else
    value = get(c,'count');
    % increment the counter
    put(c,'count', value+1);
end
x = get(c,'count');
```

- 3 Test the counter.

```
for i = 1:5
    y(i) = myCounter('myCache','myRedisConnection');
```

```

end
y
y =
    0     1     2     3     4

```

### Deployment Workflow

Before you deploy code that uses persistence to a server instance, start the persistence service and attach it to the server instance. You can start the persistence service from the system command line using `mps - cache` or follow the steps in the dashboard. This example assumes your server instance uses the default host and port: `localhost:9910`.

- 1 Package the file `myCounter.m` using the **Production Server Compiler** app or `mcc`.
- 2 Deploy the archive (`myCounter.ctf` file) to the server.
- 3 Test the counter. You can make calls to the server using the “RESTful API” from the MATLAB desktop.

```

rhs = {'myCache'}, {'myRedisConnection'};
body = mps.json.encoderrequest(rhs, 'Nargout', 1);

options = weboptions;
options.ContentType = 'text';
options.MediaType = 'application/json';
options.Timeout = 30;

for i = 1:5
    response = webwrite('http://localhost:9910/myCounter/myCounter', body, options);
    x(i) = mps.json.decoderesponse(response);
end
x = [x{:}]

x =
    0     1     2     3     4

```

As expected, the results from the testing environment workflow and the deployment environment workflow are the same.

### See Also

`get` | `mps.cache.Controller` | `mps.cache.DataCache` | `mps.cache.connect` |  
`mps.cache.control` | `mps.sync.TimedMATFileMutex` | `mps.sync.TimedRedisMutex` |  
`mps.sync.mutex` | `put`

### More About

- “Example: Calculate the Shortest Route Between Cities Using Persistence” on page 7-5



## Example: Calculate the Shortest Route Between Cities Using Persistence

This example shows how to manage persistent data in application archives deployed to MATLAB Production Server. It uses the MATLAB Production Server “RESTful API” and JSON to connect one or more instances of a MATLAB app to an archive deployed on the server.

MATLAB Production Server workers are stateless. Persistence provides a mechanism to maintain state by caching data between multiple calls to MATLAB code deployed on the server. Multiple workers have access to the cached data.

The example describes two workflows.

- 1 A testing workflow for testing the functionality of the application in a MATLAB desktop environment before deploying it to the server.
- 2 A deployment workflow that uses an active MATLAB Production Server instance to deploy the archive.

To demonstrate how to use persistence, this example uses the traveling salesman problem, which involves finding the shortest possible route between cities. This implementation stores a persistent MATLAB graph object in the data cache. Cities form the nodes of the graph and the distances between the cities form the weights associated with the graph edges. In this example, the graph is a complete graph. The testing workflow uses the local version of the route-finding functions. The deployment workflow uses route-finding-functions that are packaged into an archive and deployed to the server. The MATLAB app calls the route-finding functions. These functions read from and write graph data to the cache.

The code for the example is located at `$MPS_INSTALL/client/matlab/examples/persistence/TravelingSalesman`, where `$MPS_INSTALL` is the location where MATLAB Production Server is installed.

To host a deployable archive created with the **Production Server Compiler** app, you must have a version of MATLAB Runtime installed that is compatible with the version of MATLAB you use to create your archive. For more information, see “Supported MATLAB Runtime Versions”.

1. “Step 1: Write MATLAB Code that uses Persistence Functions” on page 7-5
2. “Step 2: Run Example in Testing Workflow” on page 7-9
3. “Step 3: Run Example in Deployment Workflow” on page 7-10

### Step 1: Write MATLAB Code that uses Persistence Functions

- 1 Write a function to initialize persistent data

Write a function to check whether a graph of cities and distances exists in the data cache. If the graph does not exist, create it from an Excel spreadsheet that contains the distance data and write it to the cache. Because only one MATLAB Production Server worker at a time can perform this write operation, use a synchronization lock to ensure that data initialization happens only once.

Connect to the cache that stores the distance data or create it if it does not exist using `mps.cache.connect`. Acquire a lock on a mutex using `mps.sync.mutex` for the duration of the write operation. Release the lock once the data is written to the cache.

Initialize the distance data using the `loadDistanceData` function.

```
function tf = loadDistanceData(connectionName, cacheName)
    c = mps.cache.connect(cacheName, 'Connection', connectionName);
    tries = 0;

    while isKey(c, 'Distances') == false && tries < 6
        lk = mps.sync.mutex('DistanceData', 'Connection', connectionName);
        if acquire(lk, 10)
            if isKey(c, 'Distances') == false
                g = initDistanceData('Distances.xlsx');
                c.Distances = g;
            end
            release(lk);
        end
        tries = tries + 1;
    end
    tf = isKey(c, 'Distances');
end
```

## 2 Write functions to read persistent data

Write a function to read the distance data graph from the data cache. Because reading data from the cache is an idempotent operation, you do not need to use synchronization locks. Connect to the cache using `mps.cache.connect` and then retrieve the graph.

Read the graph from the cache and convert it into a cell array using the `listDestinations` function.

Calculate the shortest possible route using the `findRoute` function. Use the nearest neighbor algorithm, by starting at a given city and repeatedly visiting the next nearest city until all cities have been visited.

```
function destinations = listDestinations()
    c = mps.cache.connect('TravelingSalesman', 'Connection', 'ScratchPad');
    if loadDistanceData('ScratchPad', 'TravelingSalesman') == false
        error('Failed to load distance data. Cannot continue.');
```

```
    end

    g = c.Distances;
    destinations = table2array(g.Nodes);
end

function [route, distance] = findRoute(start, destinations)
    c = mps.cache.connect('TravelingSalesman', 'Connection', 'ScratchPad');
    if loadDistanceData('ScratchPad', 'TravelingSalesman') == false
        error('Failed to load distance data. Cannot continue.');
```

```
    end

    g = c.Distances;
    route = {start};
    distance = 0;
    current = start;

    while ~isempty(destinations)
        minDistance = Inf;
        nextSegment = {};
        for n = 1:numel(destinations)
```

```

        [p,d] = shortestpath(g,current,destinations{n});
        if d < minDistance
            nextSegment = p(2:end);
            minDistance = d;
        end
    end

    current = nextSegment{end};
    distance = distance + minDistance;
    destinations = setdiff(destinations,current);
    route = [ route nextSegment ];
end
end

```

### 3 Write a function to modify persistent data

Write a function to add a new city. Adding a city modifies the graph stored in the data cache. Because this operation requires writing to the cache, use the `mps.sync.mutex` function described in Step 1 for locking. After adding a city, check that the graph is still complete by confirming that the distance between every pair of cities is known.

Add a city using the `addDestination` function. Adding a city adds a new graph node name along with new edges connecting this node to all existing nodes in the graph. The weights of the newly added edges are given by the vector `distances`. `destinations` is a cell array of character vectors that has the names of other cities in the graph.

```

function count = addDestination(name, destinations, distances)
    count = 0;
    c = mps.cache.connect('TravelingSalesman','Connection','ScratchPad');
    if loadDistanceData('ScratchPad','TravelingSalesman') == false
        error('Failed to load distance data. Cannot continue.');
```

```

    end

    lk = mps.sync.mutex('DistanceData','Connection','ScratchPad');
    if acquire(lk,10)
        g = c.Distances;
        newDestinations = setdiff(g.Nodes.Name, destinations);
        if ~isempty(newDestinations)
            error('MPS:Example:TSP:MissingDestinations', ...
                'Add distances for missing destinations: %s', ...
                strjoin(newDestinations, ', '));
        end

        src = repmat({name},1,numel(destinations));
        g = addedge(g, src, destinations, distances);
        c.Distances = g;
        release(lk);
        count = numnodes(g);
    end
end

```

### 4 Write a MATLAB app to call route-finding functions

Write a MATLAB app that wraps the functions described in Steps 2 and 3 in their respective proxy functions. The app allows you to specify a host and a port. For testing, invoke the local version of the route-finding functions when the host is blank and the port has the value 0. For the deployment workflow, invoke the deployed functions on the server running on the specified host and port. Use the `webwrite` function to send HTTP POST requests to the server.

For more information on how to write an app, see “Create and Run a Simple App Using App Designer” (MATLAB).

Write the proxy functions `findRouteProxy`, `addDestinationProxy`, and `listDestinationProxy` for the `findRoute`, `addDestination`, and `listDestination` functions, respectively.

```
function destinations = listDestinationsProxy(app)
    if isempty(app.HostEditField.Value) && ...
        app.PortEditField.Value <= 0
        destinations = listDestinations();
    return;
end

listDestinations_OPTIONS = weboptions('MediaType','application/json','Timeout',60);
listDestinations_HOST = app.HostEditField.Value;
listDestinations_PORT = app.PortEditField.Value;
noInputJSON = '{ "rhs": [], "nargout": 1 }';
destinations_JSON = ...
webwrite(sprintf('http://%s:%d/TravelingSalesman/listDestinations',listDestinations_HOST,listDestinations_PORT),noInputJSON);
if iscolumn(destinations_JSON), destinations_JSON = destinations_JSON'; end
destinations_RESPONSE = mps.json.decoderesponse(destinations_JSON);
if isstruct(destinations_RESPONSE)
    error(destinations_RESPONSE.id,destinations_RESPONSE.message);
else
    if nargout > 0, destinations = destinations_RESPONSE{1}; end
end
end

function [route,distance] = findRouteProxy(app,start,destinations)
    if isempty(app.HostEditField.Value) && ...
        app.PortEditField.Value <= 0
        [route,distance] = findRoute(start,destinations);
    return;
end
findRoute_OPTIONS = weboptions('MediaType','application/json','Timeout',60,'ContentType','application/json');
findRoute_HOST = app.HostEditField.Value;
findRoute_PORT = app.PortEditField.Value;
start_destinations_DATA = {};
if nargin > 0, start_destinations_DATA = [ start_destinations_DATA { start } ]; end
if nargin > 1, start_destinations_DATA = [ start_destinations_DATA { destinations } ]; end
route_distance_JSON = ...
webwrite(sprintf('http://%s:%d/TravelingSalesman/findRoute',findRoute_HOST,findRoute_PORT),start_destinations_DATA);
if iscolumn(route_distance_JSON), route_distance_JSON = route_distance_JSON'; end
route_distance_RESPONSE = mps.json.decoderesponse(route_distance_JSON);
if isstruct(route_distance_RESPONSE)
    error(route_distance_RESPONSE.id,route_distance_RESPONSE.message);
else
    if nargout > 0, route = route_distance_RESPONSE{1}; end
    if nargout > 1, distance = route_distance_RESPONSE{2}; end
end
end

function count = addDestinationProxy(app, name, destinations,distances)
    if isempty(app.HostEditField.Value) && ...
        app.PortEditField.Value <= 0
        count = addDestination(name, destinations,distances);
    return;
end
```

```

end

addDestination_OPTIONS = weboptions('MediaType','application/json','Timeout',60,'
addDestination_HOST = app.HostEditField.Value;
addDestination_PORT = app.PortEditField.Value;
name_destinations_distances_DATA = {};
if nargin > 0, name_destinations_distances_DATA = [ name_destinations_distances_D
if nargin > 1, name_destinations_distances_DATA = [ name_destinations_distances_D
if nargin > 2, name_destinations_distances_DATA = [ name_destinations_distances_D
count_JSON = ...
    webwrite(sprintf('http://%s:%d/TravelingSalesman/addDestination',addDestinati
if iscolumn(count_JSON), count_JSON = count_JSON; end
count_RESPONSE = mps.json.decoderesponse(count_JSON);
if isstruct(count_RESPONSE)
    error(count_RESPONSE.id,count_RESPONSE.message);
else
    if nargout > 0, count = count_RESPONSE{1}; end
end
end
end

```

## Step 2: Run Example in Testing Workflow

Test the example code in the MATLAB desktop environment. To do so, copy the all the files located at `$MPS_INSTALL/client/matlab/examples/persistence/TravelingSalesman` to a writable folder on your system, for example, `/tmp/persistence_example`. Start the MATLAB desktop and set the current working directory to `/tmp/persistence_example` using the `cd` command.

For testing purposes, control a persistence service from the MATLAB desktop with the `mps.cache.control` function. This function returns an `mps.cache.Controller` object that manages the life cycle of a local persistence service.

- 1 Create an `mps.cache.Controller` object for a local persistence service that uses the Redis persistence provider.

```
>> ctrl = mps.cache.control('ScratchPad', 'Redis', 'Port', 8675);
```

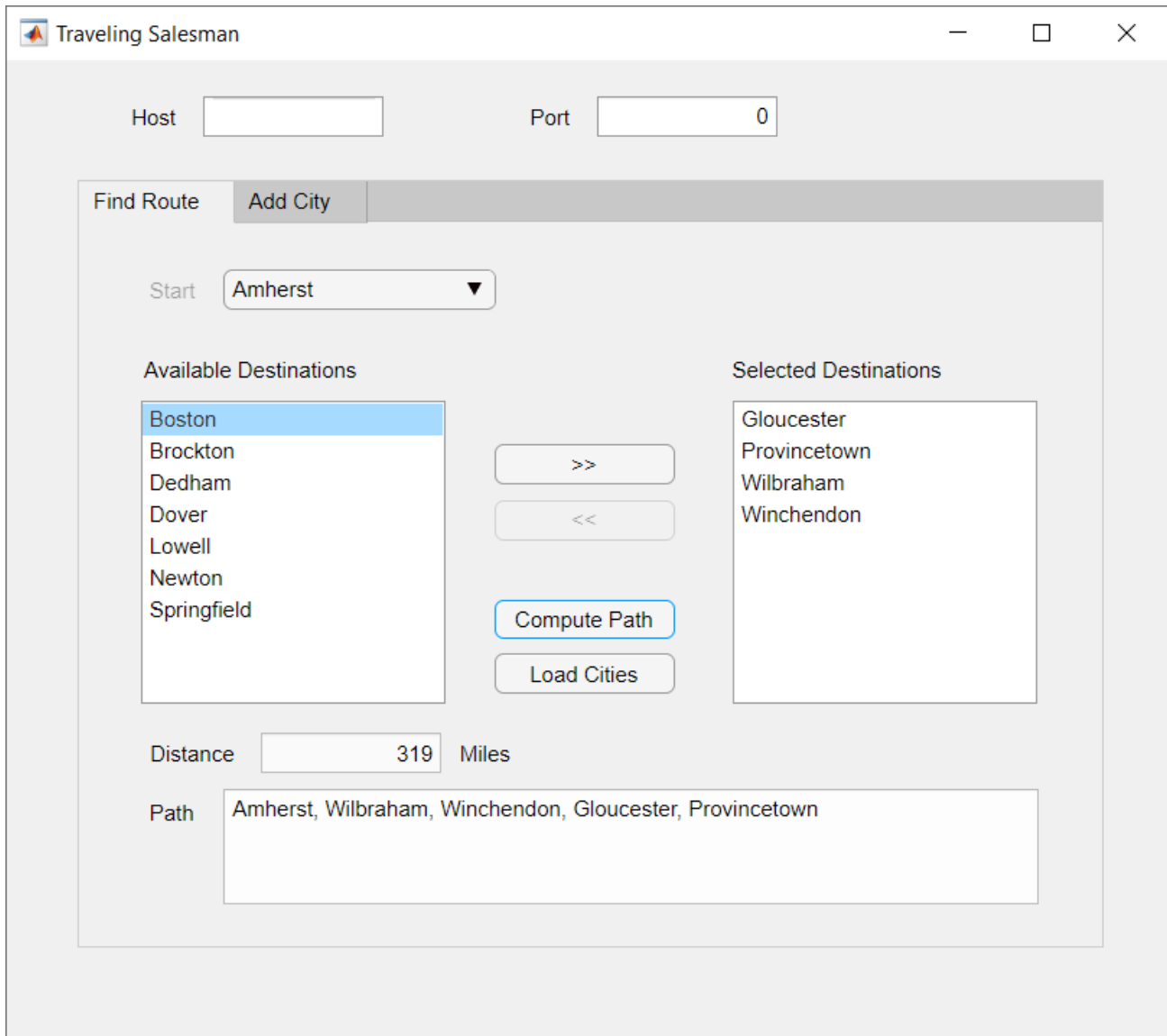
When active, this controller enables a connection named `ScratchPad`. Connection names link caches to storage locations in persistence services. The `mps.cache.connect` function requires connection names to create data caches. The MATLAB Production Server administrator sets connection names in the cache configuration file `mps_cache_config`. By using the same connection names in MATLAB desktop sessions, you enable your code to move from development through testing to production without change.

- 2 Start the persistence service using `start`.
- 3 Start the `TravelingSalesman` route-finding app that uses the persistence service.

```
>> TravelingSalesman
```

The app starts with default values for **Host** and **Port**.

Click **Load Cities** to load the list of cities. Use the **Start** menu to set a starting location and the **>>** and **<<** buttons to select and deselect cities to visit. Click **Compute Path** to display a route that visits all the cities.



- 4 When you close the app, stop the persistence service using `stop`. Stopping a persistence service will delete the data stored by that service.

```
>> stop(ctrl);
```

### Step 3: Run Example in Deployment Workflow

To run the example in the deployment workflow, copy all the files located at `$MPS_INSTALL/client/matlab/examples/persistence/TravelingSalesman` to a writeable folder on your system, for example, `/tmp/persistence_example`. Start the MATLAB desktop and set the current working directory to `/tmp/persistence_example` using the MATLAB `cd` command.

The deployment workflow manages the lifetime of a persistence service outside of a MATLAB desktop environment and invokes the route-finding functions packaged in an archive deployed to the server.

- 1 Create a MATLAB Production Server instance

Create a server from the system command line using `mps -new`. For more information, see “Create Server Instance”. If you have not already set up your server environment, see `mps -setup` for more information.

Create a new server `server_1` located in the folder `tmp`.

```
mps-new /tmp/server_1
```

Alternatively, use the MATLAB Production Server dashboard to create a server. For more information, see “Set Up and Log In to MATLAB Production Server Dashboard”.

## 2 Create a persistence service connection

The deployable archive requires a persistence service connection named `ScratchPad`. Use the dashboard to create the `ScratchPad` connection or copy the file `mps_cache_config` from the example directory to the config directory of your server instance. If you already have an `mps_cache_config` file in your config directory, edit it to add the `ScratchPad` connection as specified in the example `mps_cache_config`.

## 3 Create a deployable archive with the Production Server Compiler App and deploy it to the server

### 1 Open **Production Server Compiler** app

- MATLAB toolstrip: On the **Apps** tab, under **Application Deployment**, click **Production Server Compiler**.

- MATLAB command prompt: Enter `productionServerCompiler`.

### 2 In the **Application Type** menu, select **Deployable Archive**.

### 3 In the **Exported Functions** field, add `findRoute.m`, `listDestinations.m` and `addDestination.m`.

### 4 Under **Archive information**, rename the archive to `TravelingSalesman`.

### 5 Under **Additional files required for your archive to run**, add `Distances.xlsx`.

### 6 Click **Package**.

### 7 The generated deployable archive `TravelingSalesman.ctf` is located in the `for_redistribution` folder of the project. Copy the `TravelingSalesman.ctf` file to the `auto_deploy` folder of the server, `/tmp/server_1/auto_deploy` in this example, for hosting.

## 4 Start the server instance

Start the server from the system command line using `mps -start`.

```
mps-start -C /tmp/server_1
```

Alternatively, use the dashboard to start the server.

## 5 Start the persistence service

Start the persistence service from the system command line using `mps -cache`.

```
mps-cache start -C /tmp/server_1 --connection ScratchPad
```

Alternatively, use the dashboard to start and attach the persistence service.

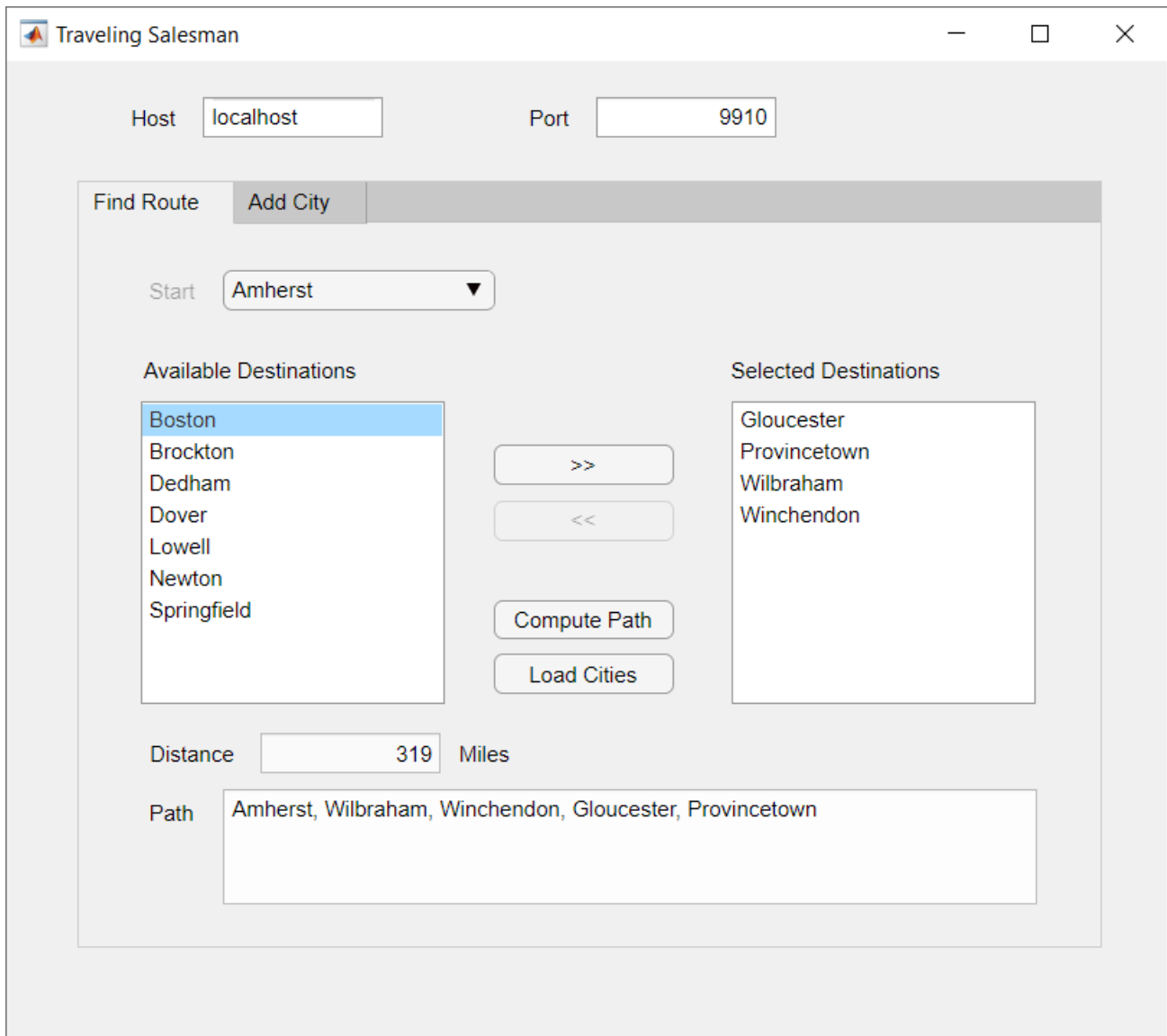
## 6 Test the app

Start the `TravelingSalesman` route-finding app that uses the persistence service.

```
>> TravelingSalesman
```

The app starts with empty values for **Host** and **Port**. Refer to the server configuration file `main_config` located at `server_name/config` to get the host and port values for your MATLAB Production Server instance. For this example, find the config file at `/tmp/server_1/config`. Enter the host and port values in the app.

Click **Load Cities** to load the list of cities. Use the **Start** menu to set a starting location and the **>>** and **<<** buttons to select and deselect cities to visit. Click **Compute Path** to display a route that visits all the cities.



The results from the testing environment workflow and the deployment environment workflow are the same.

### See Also

`mps.cache.Controller` | `mps.cache.DataCache` | `mps.cache.connect` | `mps.cache.control` | `mps.sync.TimedMATFileMutex` | `mps.sync.TimedRedisMutex` | `mps.sync.mutex`



## **More About**

- “Use a Data Cache to Persist Data” on page 7-2



# Persistence Functions

---

## mps.cache.DataCache

Represent cache concept in MATLAB code

### Description

`mps.cache.DataCache` represents the concept of cache in MATLAB code. It is an abstract class that serves as a superclass for each persistence provider-specific data cache class.

Currently, Redis and MATLAB are the only supported persistence providers. Therefore, the cache objects will be of type `mps.cache.RedisCache` or `mps.cache.MATFileCache`.

### Creation

Create a persistence provider-specific subclass of `mps.cache.DataCache` using `mps.cache.connect`.

### Properties

See provider-specific subclasses for properties.

### Object Functions

<code>mps.cache.connect</code>	Connect to cache, or create a cache if it doesn't exist
<code>bytes</code>	Return the number of bytes of storage used by value stored at each key
<code>clear</code>	Remove all keys and values from cache
<code>flush</code>	Write all locally modified keys to the persistence service
<code>get</code>	Fetch values of keys from cache
<code>getp</code>	Get the value of a public cache property
<code>isKey</code>	Determine if the cache contains specified keys
<code>keys</code>	Get all keys from cache
<code>length</code>	Number of key-value pairs in the data cache
<code>purge</code>	Flush all local data to the persistence service
<code>put</code>	Write key-value pairs to cache
<code>remove</code>	Remove keys from cache
<code>retain</code>	Store remote keys from cache locally or return locally stored keys

### Examples

#### Connect to a Redis Cache

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519);
start(ctrl)
c = mps.cache.connect('myCache', 'Connection', 'myRedisConnection')
```

c =

RedisCache with properties:

```
    Host: 'localhost'  
    Port: 4519  
    Name: 'myCache'  
Operations: "read | write | create | update"  
    LocalKeys: {}  
    Connection: 'myRedisConnection'
```

Use `getp` instead of dot notation to access properties.

## See Also

`mps.cache.Controller`

## Topics

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

## mps.cache.Controller

Manage the life cycle of a persistence service in a MATLAB testing environment

### Description

`mps.cache.Controller` is used to manage the life cycle of a persistence service in a MATLAB testing environment. You can perform various actions such as starting and stopping the service using the object.

### Creation

Create a `mps.cache.Controller` object using `mps.cache.control`.

### Properties

#### ActiveConnection — Connection indicator

True | False

This property is read-only.

Indicates whether the connection to the persistence provider is active or not. The value is `True` when the persistence service is attached to the MATLAB session, otherwise it is `False`.

Example: `ActiveConnection: False`

#### ManageService — Service management indicator

True | False | Unknown

This property is read-only.

Indicates whether the controller object is managing the persistence service or not. `ManageService` is `True` if the persistence service is started using the controller's `startstart` method and `False` if the MATLAB session is attached to the persistence service using the controller's `attach` method. In all other cases, the value is set to `Unknown`.

If `ManageService` is `True`, destroying the controller object via `delete` or exiting MATLAB will stop the persistence service.

Example: `ManageService: True`

#### Host — Host name

character vector

This property is read-only.

Name of the system hosting the persistence service.

This property is not displayed when you create a controller that uses MATLAB as a persistence provider.

Example: Host: 'localhost'

### **Port — Port number**

positive scalar

This property is read-only.

Port number for persistence service.

This property is not displayed when you create a controller that uses MATLAB as a persistence provider.

Example: Port: 4519

### **ProviderName — Name of persistence provider**

'Redis' | 'MatlabTest'

This property is read-only.

Name of the persistence provider.

Currently, Redis is the only supported persistence provider.

You can also use MATLAB as a persistence provider for testing purposes. If you use MATLAB as a persistence provider, the provider name is displayed as 'MatlabTest'.

Example: ProviderName: 'Redis'

Example: ProviderName: 'MatlabTest'

### **ConnectionName — Name of connection**

character vector | string

This property is read-only.

Name of connection to persistence service.

Example: ConnectionName: 'myRedisConnection'

### **Folder\* — Storage folder path**

character vector

This property is read-only.

Storage folder path. The folder displayed is used as a database.

\* This property is displayed only when you create a controller that uses MATLAB as a persistence provider.

Example: Folder: 'c:\tmp'

## **Object Functions**

mps.cache.control	Create a persistence service controller object
start	Start a persistence service and attach it a to MATLAB session
stop	Stop a persistence service and detach it from a MATLAB session
restart	Restart a persistence service and attach it to a MATLAB session

attach	Connect a MATLAB session to a persistence service that is already running
detach	Disconnect MATLAB session from a persistence service that is already running
ping	Test whether the persistence service is reachable
version	Version number for persistence provider

## Examples

### Create a Redis Service Controller

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519)
```

```
ctrl =
```

```
Controller with properties:
```

```
ActiveConnection: False
ManageService: Unknown
Host: 'localhost'
Port: 4519
Operations: "read | write | create | update"
ProviderName: 'Redis'
ConnectionName: 'myRedisConnection'
```

### Create a MATLAB Service Controller

```
mctrl = mps.cache.control('myMATFileConnection', 'MatlabTest', 'Folder', 'c:\tmp')
```

```
mctrl =
```

```
Controller with properties:
```

```
ActiveConnection: False
ManageService: Unknown
Folder: 'c:\tmp'
Operations: "read | write | create | update"
ProviderName: 'MatlabTest'
ConnectionName: 'myMATFileConnection'
```

## See Also

`mps.cache.DataCache`

### Topics

“Use a Data Cache to Persist Data” on page 7-2

### Introduced in R2018b



# mps.cache.connect

Connect to cache, or create a cache if it doesn't exist

## Syntax

```
c = mps.cache.connect(cacheName)
c = mps.cache.connect(cacheName, 'Connection', connectionName)
```

## Description

`c = mps.cache.connect(cacheName)` connects to a cache when there's a single connection to a persistence service.

`c = mps.cache.connect(cacheName, 'Connection', connectionName)` connects to a cache using the connection specified by `connectionName` when there are multiple connections to a persistence service.

## Examples

### Create a Cache When There is a Single Connection to a Persistence Service

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

When you have a single connection, you do not need to specify the connection name to `mps.cache.connect`.

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519)
start(ctrl)
c = mps.cache.connect('myCache');

c =
```

RedisCache with properties:

```
Host: 'localhost'
Port: 4519
Name: 'myCache'
Operations: "read | write | create | update"
LocalKeys: {}
Connection: 'myRedisConnection'
```

Use `getp` instead of dot notation to access properties.

### Create a Cache When There are Multiple Connections to a Persistence Service

When you have multiple connections to a persistence service, create a cache by specifying the connection name associated with the service you want to use.

```
ctrl_1 = mps.cache.control('myRedisConnection1', 'Redis', 'Port', 4519)
start(ctrl_1)
ctrl_2 = mps.cache.control('myRedisConnection2', 'Redis', 'Port', 4520)
start(ctrl_2)
c = mps.cache.connect('myCache', 'Connection', 'myRedisConnection1')
```

```
c =
```

RedisCache with properties:

```
    Host: 'localhost'
    Port: 4519
    Name: 'myCache'
Operations: "read | write | create | update"
  LocalKeys: {}
  Connection: 'myRedisConnection1'
```

Use `getp` instead of dot notation to access properties.

## Input Arguments

### **cacheName** — Cache name to connect to or create

character vector

Cache name to connect to or create, specified as a character vector.

Example: 'myCache'

### **connectionName** — Name of connection

character vector

Name of connection to persistence service, specified as a character vector.

Example: 'Connection', 'myRedisConnection'

## Output Arguments

### **c** — Data cache object

persistence provider-specific data cache object

A persistence provider specific data cache object.

Currently, Redis and MATLAB are the only supported persistence providers. Therefore, the cache objects will be of type `mps.cache.RedisCache` or `mps.cache.MATFileCache`.

## See Also

`mps.cache.DataCache`

**Introduced in R2018b**

# mps.cache.control

Create a persistence service controller object

## Syntax

```
ctrl = mps.cache.control(connectionName,Provider,'Port',num)
ctrl = mps.cache.control(connectionName,Provider,'Folder',folderPath)
```

## Description

`ctrl = mps.cache.control(connectionName,Provider,'Port',num)` creates a persistence service controller object using a connection to a persistence service specified by `connectionName`, a persistence provider specified by `Provider`, and a port number `num` for the service.

You cannot compile and deploy this function on the server. This function is available only for testing.

`ctrl = mps.cache.control(connectionName,Provider,'Folder',folderPath)` creates a persistence service controller object that uses a folder specified by `folderPath` as a database.

Use this syntax when you want to use MATLAB as a persistence provider for testing purposes.

You cannot compile and deploy this function on the server. This function is available only for testing.

## Examples

### Create a Redis Service Controller

```
ctrl = mps.cache.control('myRedisConnection','Redis','Port',4519)
```

```
ctrl =
```

Controller with properties:

```
ActiveConnection: False
  ManageService: Unknown
    Host: 'localhost'
    Port: 4519
    Operations: "read | write | create | update"
  ProviderName: 'Redis'
  ConnectionName: 'myRedisConnection'
```

### Create a MATLAB Service Controller

```
mctrl = mps.cache.control('myMATFileConnection','MatlabTest','Folder','c:\tmp')
```

```
mctrl =
```

Controller with properties:

```
ActiveConnection: False
  ManageService: Unknown
    Folder: 'c:\tmp'
```

```
Operations: "read | write | create | update"  
ProviderName: 'MatlabTest'  
ConnectionName: 'myMATFileConnection'
```

## Input Arguments

### **connectionName** — Name of the connection

character vector | string

Name of the connection to the persistence service, specified as a character vector.

The `connectionName` links a MATLAB session to a persistence service.

Example: 'myRedisConnection'

### **Provider** — Name of the persistence provider

'Redis' | 'MatlabTest'

Name of the persistence provider, specified as a character vector.

You can use MATLAB as a persistence provider for testing purposes. If you use MATLAB as a persistence provider, specify the provider name as 'MatlabTest'.

Example: 'Redis'

Example: 'MatlabTest'

### **num** — Port number

positive scalar

Port number for the persistence service.

Example: 'Port', 4519

### **folderPath** — Storage folder path

character vector

Storage folder path, specified as a character vector.

Specify this input only when you want to use MATLAB as a persistence provider for testing purposes. A folder specified by `folderPath` serves as a database.

Example: 'Folder', 'c:\tmp'

## Output Arguments

### **ctrl** — Persistence provider service controller object

`mps.cache.Controller` object

Persistence provider service controller returned as a `mps.cache.Controller` object.

## See Also

`mps.cache.Controller` | `restart` | `start` | `stop`

### Topics

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

## attach

Connect a MATLAB session to a persistence service that is already running

### Syntax

```
attach(ctrl)
```

### Description

`attach(ctrl)` connects a MATLAB session to a persistence service that is already running.

### Examples

#### Connect a MATLAB Session to a Persistence Service

Attach MATLAB code to a persistence service.

Start a persistence service outside your MATLAB session from system command line using `mps-cache` or using the dashboard. Assuming you started the service using a connection name `myOutsideRedisConnection` at port `8899`, attach your MATLAB session to it from the MATLAB desktop.

```
ctrl = mps.cache.control('myOutsideRedisConnection','Redis','Port',8899);  
attach(ctrl)
```

### Input Arguments

#### **ctrl** — Service controller

`mps.cache.Controller` object

Persistence service controller, represented as a `mps.cache.Controller` object.

Example: `attach(ctrl)`

### See Also

`detach` | `restart` | `start` | `stop`

### Topics

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

# detach

Disconnect MATLAB session from a persistence service that is already running

## Syntax

```
detach(ctrl)
```

## Description

`detach(ctrl)` disconnects MATLAB session from a persistence service that is already running.

## Examples

### Disconnect MATLAB Code

Disconnect MATLAB code from a persistence service.

First, create a persistence service controller object and use that object to start the persistence service. Once you have a persistence service running, you can connect MATLAB code to it. You can then disconnect the code from the service.

```
ctrl = mps.cache.control('myRedisConnection','Redis','Port',4519);  
start(ctrl)  
attach(ctrl)  
detach(ctrl)
```

## Input Arguments

### `ctrl` — Service controller

`mps.cache.Controller` object

Persistence service controller, represented as a `mps.cache.Controller` object.

Example: `detach(ctrl)`

## See Also

`attach` | `restart` | `start` | `stop`

## Topics

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

## start

Start a persistence service and attach it to a MATLAB session

### Syntax

```
start(ctrl)
```

### Description

`start(ctrl)` starts a persistence service represented by `ctrl` and attaches it to a current MATLAB session.

- To make a persistence service available in a MATLAB session, the service must be started and then attached to the MATLAB session. `start` performs both these actions.
- If a persistence service has already been started, there is no need to call `start`. Use `attach` instead.
- `start` and `stop`, `attach` and `detach` must be used in pairs.
- If you connected a persistence service to your MATLAB session with `start`, you must disconnect with `stop`.
- If you connected with `attach`, you must disconnect with `detach`.

### Examples

#### Start a Persistence Service

Start a persistence service.

First, create a persistence service controller object and use that object to start the persistence service.

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519);  
start(ctrl)
```

### Input Arguments

#### `ctrl` — Service controller

`mps.cache.Controller` object

Persistence service controller, represented as a `mps.cache.Controller` object.

Example: `start(ctrl)`

### See Also

`attach` | `detach` | `restart` | `stop`

#### Topics

“Use a Data Cache to Persist Data” on page 7-2



**Introduced in R2018b**

## stop

Stop a persistence service and detach it from a MATLAB session

### Syntax

```
stop(ctrl)
```

### Description

`stop(ctrl)` stops a persistence service represented by `ctrl` and detaches it from a current MATLAB session.

- You cannot stop a service that has not been started.
- You can only stop a service that has been started using `start`.
- Exiting MATLAB will automatically call `stop` on all persistence services that were started using `start`.

### Examples

#### Stop a Persistence Service

Stop a persistence service.

First, create a persistence service controller object and use that object to start the persistence service. Once you have a persistence service running, you can then stop it.

```
ctrl = mps.cache.control('myRedisConnection','Redis','Port',4519);  
start(ctrl)  
stop(ctrl)
```

### Input Arguments

#### **ctrl** — Service controller

`mps.cache.Controller` object

Persistence service controller, represented as a `mps.cache.Controller` object.

Example: `stop(ctrl)`

### See Also

`attach` | `detach` | `restart` | `start`

#### Topics

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

# restart

Restart a persistence service and attach it to a MATLAB session

## Syntax

```
restart(ctrl)
```

## Description

`restart(ctrl)` restarts a persistence service represented by `ctrl`. You only restart a services you originally started using `start`.

## Examples

### Restart a Persistence Provider

Restart a persistence service.

First, create a persistence service controller object and use that object to start the persistence service. Once you have a persistence service running, you can then restart it.

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519);  
start(ctrl)  
restart(ctrl)
```

## Input Arguments

### `ctrl` — Service controller

`mps.cache.Controller` object

Persistence service controller, represented as a `mps.cache.Controller` object.

Example: `restart(ctrl)`

## See Also

`attach` | `detach` | `start` | `stop`

## Topics

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

## ping

Test whether the persistence service is reachable

### Syntax

```
ping(ctrl)
```

### Description

`ping(ctrl)` tests whether the persistence service is reachable. In order to ping a persistence service, it must be started and attached to your MATLAB session.

### Examples

#### Ping Persistence Service

Test whether the persistence service is reachable.

First, create a persistence service controller object and use that object to start the persistence service. Once you have a persistence service running, you can ping the service.

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519);  
start(ctrl)  
ping(ctrl)
```

```
Sending ping to Redis on localhost:4519.  
Redis service running on localhost:4519.
```

```
ans =
```

```
    logical
```

```
    1
```

### Input Arguments

#### **ctrl** — Service controller

`mps.cache.Controller` object

Persistence service controller, represented as a `mps.cache.Controller` object.

Example: `ping(ctrl)`

### See Also

`restart` | `start` | `stop`

### Topics

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

## version

Version number for persistence provider

### Syntax

```
version(ctrl)
```

### Description

`version(ctrl)` returns the version number for the persistence provider. In order to get the version number of the persistence provider, the persistence service must be started and attached to your MATLAB session.

### Examples

#### Get Version Number

Get the version number of the persistence provider that the persistence service is connected to.

First, create a persistence service controller object and use that object to start the persistence service. Once you have a persistence service running, you can get the version number.

```
ctrl = mps.cache.control('myRedisConnection','Redis','Port',4519);  
start(ctrl)  
version(ctrl)
```

```
Redis version: 3.0.504
```

### Input Arguments

#### **ctrl** — Service controller

`mps.cache.Controller` object

Persistence service controller, represented as a `mps.cache.Controller` object.

Example: `version(ctrl)`

### See Also

`restart` | `start` | `stop`

### Topics

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

## bytes

Return the number of bytes of storage used by value stored at each key

### Syntax

```
b = bytes(c,keys)
```

### Description

`b = bytes(c,keys)` returns the number of bytes of storage used by value stored at each key.

### Examples

#### Get the Number of Bytes of Storage Used by a Value in the Cache

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection','Redis','Port',4519);
start(ctrl)
c = mps.cache.connect('myCache','Connection','myRedisConnection');
```

Add keys and values to the cache and then get the number of bytes of storage used by a value stored at each key in the cache. Represent the keys and the bytes used by each value of key as a MATLAB table.

```
put(c,'keyOne',10,'keyTwo',20,'keyThree',30,'keyFour',[400 500],'keyFive',magic(5))
b = bytes(c,{'keyOne','keyTwo','keyThree','keyFour','keyFive'})
tt = table(keys(c), bytes(c,keys(c)),'VariableNames',{'Keys','Bytes'})
```

b =

```
    72    72    72    80   264
```

tt =

5×2 table

Keys	Bytes
'keyFive'	264
'keyFour'	80
'keyOne'	72
'keyThree'	72
'keyTwo'	72

### Input Arguments

#### c — Data cache

persistence provider specific data cache object

A data cache represented by a persistence provider specific data cache object.

Currently, Redis and MATLAB are the only supported persistence providers. Therefore, the cache objects will be of type `mps.cache.RedisCache` or `mps.cache.MATFileCache`.

Example: `c`

**keys — Keys**

cell array of character vectors

A list of all the keys, specified as a cell array of character vectors.

Example: `{'keyOne', 'keyTwo', 'keyThree', 'keyFour', 'keyFive'}`

**Output Arguments****b — Number of bytes**

numeric row vector

Number of bytes used by each value associated with a key, returned as a numeric row vector.

The byte counts in the output vector appear in the same order as the corresponding input keys. `b(i)` is the byte count for `keys(i)`.

**See Also**

`get` | `keys` | `length` | `put`

**Topics**

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**



# clear

Remove all keys and values from cache

## Syntax

```
n = clear(c)
```

## Description

`n = clear(c)` removes all keys and values from cache and returns the number of keys cleared from the cache in `n`.

`clear` removes both local and remote keys and values.

## Examples

### Clear All Keys and Values from Cache

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection','Redis','Port',4519);
start(ctrl)
c = mps.cache.connect('myCache','Connection','myRedisConnection');
```

Add keys and values to the cache and display them as a MATLAB table.

```
put(c,'keyOne',10,'keyTwo',20,'keyThree',30,'keyFour',[400 500],'keyFive',magic(5))
tt = table(keys(c), get(c,keys(c)),'VariableNames',{'Keys','Values'})
```

```
tt =
```

```
5x2 table
```

Keys	Values
'keyFive'	[5x5 double]
'keyFour'	[1x2 double]
'keyOne'	[ 10]
'keyThree'	[ 30]
'keyTwo'	[ 20]

Clear the cache and check if it is empty.

```
n = clear(c)
k = keys(c)
```

```
n =
```

```
int64
```

5

k =

0×1 empty cell array

## Input Arguments

### **c** — Data cache

persistence provider specific data cache object

A data cache represented by a persistence provider specific data cache object.

Currently, Redis and MATLAB are the only supported persistence providers. Therefore, the cache objects will be of type `mps.cache.RedisCache` or `mps.cache.MATFileCache`.

Example: c

## Output Arguments

### **n** — Number of key-value pairs

integer

Number of key-value pairs removed, returned as an integer.

Example: 5

## See Also

`flush` | `keys` | `purge` | `put` | `remove` | `retain`

## Topics

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

## flush

Write all locally modified keys to the persistence service

### Syntax

```
modKeys = flush(c)
```

### Description

`modKeys = flush(c)` writes all locally modified data in `c` to the persistence service and returns a list of keys that have been modified.

`flush` does not clear the list of retained keys.

### Examples

#### Write All Locally Modified Data to the Persistence Service

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519);
start(ctrl)
c = mps.cache.connect('myCache', 'Connection', 'myRedisConnection');
```

Add keys and values to the cache and display them as a MATLAB table.

```
put(c, 'keyOne', 10, 'keyTwo', 20, 'keyThree', 30, 'keyFour', [400 500], 'keyFive', magic(5))
tt = table(keys(c), get(c, keys(c)), 'VariableNames', {'Keys', 'Values'})
```

```
tt =
```

```
5×2 table
```

Keys	Values
'keyFive'	[5×5 double]
'keyFour'	[1×2 double]
'keyOne'	[ 10]
'keyThree'	[ 30]
'keyTwo'	[ 20]

Retain a single key locally and verify that it shows up as a local key in the cache object.

```
retain(c, 'keyOne')
display(c)
```

```
c =
```

RedisCache with properties:

```

    Host: 'localhost'
    Port: 4519
    Name: 'myCache'
    Operations: "read | write | create | update"
    LocalKeys: {'keyOne'}
    Connection: 'myRedisConnection'

```

Use `getp` instead of dot notation to access properties.

Modify the local key and flush it to the remote cache. Display the keys and values in the cache as a MATLAB table.

```

put(c, 'keyOne', rand(3))
modKeys = flush(c)
tt = table(keys(c), get(c,keys(c))', 'VariableNames', {'Keys', 'Values'})

```

modKeys =

1×1 cell array

```
{'keyOne'}
```

tt =

5×2 table

Keys	Values
'keyFive'	[5×5 double]
'keyFour'	[1×2 double]
'keyOne'	[3×3 double]
'keyThree'	[ 30]
'keyTwo'	[ 20]

## Input Arguments

### **c** — Data cache

persistence provider specific data cache object

A data cache represented by a persistence provider specific data cache object.

Currently, Redis and MATLAB are the only supported persistence providers. Therefore, the cache objects will be of type `mps.cache.RedisCache` or `mps.cache.MATFileCache`.

Example: c

## Output Arguments

### **modKeys** — Modified keys

cell array of character vectors

A list of the modified keys that were written to the persistence service, returned as a cell array of character vectors.

## **See Also**

clear | keys | purge | remove | retain

## **Topics**

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

## get

Fetch values of keys from cache

### Syntax

```
values = get(c,keys)
```

### Description

`values = get(c,keys)` fetches values of keys specified by `keys` from the cache specified by `c`. Values are returned in the same order as input variables as a cell array.

### Examples

#### Get Values for Keys from Cache

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection','Redis','Port',4519);
start(ctrl)
c = mps.cache.connect('myCache', 'Connection', 'myRedisConnection');
```

Add keys and values to the cache.

```
put(c,'keyOne',10,'keyTwo',20,'keyThree',30,'keyFour',[400 500],'keyFive',magic(5))
```

Get all the keys and associated values and display them as a MATLAB table.

```
k = keys(c)
v = get(c,{'keyOne','keyTwo','keyThree','keyFour','keyFive'})
tt = table(keys(c), get(c,keys(c)),'VariableNames',{'Keys','Values'})
```

k =

5×1 cell array

```
{'keyFive' }
{'keyFour' }
{'keyOne' }
{'keyThree'}
{'keyTwo' }
```

v =

1×5 cell array

```
{[10]}    {[20]}    {[30]}    {1×2 double}    {5×5 double}
```

tt =

5×2 table

Keys	Values
'keyFive'	[5×5 double]
'keyFour'	[1×2 double]
'keyOne'	[ 10]
'keyThree'	[ 30]
'keyTwo'	[ 20]

## Input Arguments

### **c** — Data cache

persistence provider specific data cache object

A data cache represented by a persistence provider specific data cache object.

Currently, Redis and MATLAB are the only supported persistence providers. Therefore, the cache objects will be of type `mps.cache.RedisCache` or `mps.cache.MATFileCache`.

Example: `c`

### **keys** — Keys

cell array of character vectors

A cell array of keys whose values you want to retrieve from cache.

Example: `{'keyOne', 'keyTwo', 'keyThree', 'keyFour', 'keyFive'}`

## Output Arguments

### **values** — Values

cell array

A list of values associated with keys, returned as a cell array.

## See Also

`getp` | `keys` | `length` | `put`

### Topics

“Use a Data Cache to Persist Data” on page 7-2

### Introduced in R2018b

## getp

Get the value of a public cache property

### Syntax

```
value = getp(c,property)
```

### Description

`value = getp(c,property)` gets the value of a public cache property.

Ordinarily, you would be able to access the public properties of a cache object using the dot notation. For example: `c.Connection`. However, all cache objects use dot reference and dot assignment to refer to keys stored in the cache rather than cache object properties. Therefore, `c.Connection` refers to a key named `Connection` in the cache instead of the cache's `Connection` property.

There is no `setp` method since all cache properties are read-only.

### Examples

#### Get the Value of a Named, Public, Hidden Property

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519);
start(ctrl)
c = mps.cache.connect('myCache', 'Connection', 'myRedisConnection');
```

Retrieve the connection name.

```
getp(c, 'Connection')
```

```
ans =
```

```
    'myRedisConnection'
```

### Input Arguments

#### **c** — Data cache

persistence provider specific data cache object

A data cache represented by a persistence provider specific data cache object.

Currently, Redis and MATLAB are the only supported persistence providers. Therefore, the cache objects will be of type `mps.cache.RedisCache` or `mps.cache.MATFileCache`.

Example: `c`



**property — Property name**

character vector

Property name, specified as a character vector. The common public cache properties are `Name`, `LocalKeys`, and `Connection`. Provider-specific cache objects may have additional properties. For example, `mps.cache.RedisCache` has the properties `Host` and `Port`.

Example: `'Connection'`

**Output Arguments****value — Property value**

valid value

A valid property value.

**See Also**

`get` | `keys` | `put`

**Topics**

"Use a Data Cache to Persist Data" on page 7-2

**Introduced in R2018b**

## isKey

Determine if the cache contains specified keys

### Syntax

```
TF = isKey(c,keys)
```

### Description

`TF = isKey(c,keys)` returns a logical 1 (`true`) if `c` contains the specified key, and returns a logical 0 (`false`) otherwise.

If `keys` is an array that specifies multiple keys, then `TF` is a logical array of the same size, and `TF{i}` is `true` if `keys{i}` exists in cache `c`.

### Examples

#### Determine if the Cache Contains Specified Keys

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection','Redis','Port',4519);
start(ctrl)
c = mps.cache.connect('myCache','Connection','myRedisConnection');
```

Add keys and values to the cache.

```
put(c,'keyOne',10,'keyTwo',20,'keyThree',30,'keyFour',[400 500],'keyFive',magic(5))
```

Determine if the cache contains specified keys.

```
TF = isKey(c,{'keyOne','keyTW00','keyTREE','key4','keyFive'})
```

```
TF =
```

```
1×5 logical array
```

```
1 0 0 0 1
```

### Input Arguments

#### **c** — Data cache

persistence provider specific data cache object

A data cache represented by a persistence provider specific data cache object.

Currently, Redis and MATLAB are the only supported persistence providers. Therefore, the cache objects will be of type `mps.cache.RedisCache` or `mps.cache.MATFileCache`.

Example: `c`

**keys — Keys to search for**

character vector | string | cell array of character vectors or strings

Keys to search for in the cache object `c`, specified as a character vector, string, or cell array of character vectors or strings. To search for multiple keys, specify `keys` as a cell array.

Example: {'keyOne', 'keyTW00', 'keyTREE', 'key4', 'keyFive'}

**Output Arguments****TF — Logical value**

logical array

A logical array of the same size as `keys` indicating which specified keys were found in the data cache. TF has a logical 1 (`true`) if `c` contains a key specified by `keys`, and a logical 0 (`false`) otherwise.

**See Also**

`get` | `keys` | `length` | `put`

**Topics**

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

## keys

Get all keys from cache

### Syntax

```
k = keys(c)
```

### Description

`k = keys(c)` returns a list of all the keys in a data cache as a cell array.

### Examples

#### Get Keys from Cache

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection','Redis','Port',4519);  
start(ctrl)  
c = mps.cache.connect('myCache', 'Connection', 'myRedisConnection');
```

Add keys and values to the cache.

```
put(c,'keyOne',10,'keyTwo',20,'keyThree',30,'keyFour',[400 500],'keyFive',magic(5))
```

Get all keys.

```
k = keys(c)
```

```
k =
```

```
5×1 cell array
```

```
{'keyFive' }  
{'keyFour' }  
{'keyOne' }  
{'keyThree'}  
{'keyTwo' }
```

### Input Arguments

#### **c** — Data cache

persistence provider specific data cache object

A data cache represented by a persistence provider specific data cache object.

Currently, Redis and MATLAB are the only supported persistence providers. Therefore, the cache objects will be of type `mps.cache.RedisCache` or `mps.cache.MATFileCache`.

Example: `c`

## Output Arguments

### **k — Keys**

cell array of character vectors

Keys from cache, returned as a cell array of character vectors.

## See Also

bytes | get | isKey | length | put

## Topics

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

## length

Number of key-value pairs in the data cache

### Syntax

```
num = length(c)
num = length(c, location)
```

### Description

`num = length(c)` returns the total number of key-value pairs in the data cache `c`.

`num = length(c, location)` returns the numbers of key-value pairs in the data cache `c` stored remotely or locally as specified by `location`.

### Examples

#### Count the Number of Key-Value Pairs

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519);
start(ctrl)
c = mps.cache.connect('myCache', 'Connection', 'myRedisConnection');
```

Retain a few keys locally.

```
retain(c, {'keyOne', 'keyTwo'})
```

Add keys and values to the cache.

```
put(c, 'keyOne', 10, 'keyTwo', 20, 'keyThree', 30, 'keyFour', [400 500], 'keyFive', magic(5))
```

Count the number of keys-value pairs.

```
numTotal = length(c)
numRemote = length(c, 'Remote')
numLocal = length(c, 'Local')
```

```
numTotal =
```

```
int64
```

```
5
```

```
numRemote =
```

```
int64
```

```
3
```

```
numLocal =  
    int64  
    2
```

Since `keyOne` and `keyTwo` were retained before being written to the cache, they were never written to the persistence service. They are stored locally until flushed or purged to the persistence service.

## Input Arguments

### **c** — Data cache

persistence provider specific data cache object

A data cache represented by a persistence provider specific data cache object.

Currently, Redis and MATLAB are the only supported persistence providers. Therefore, the cache objects will be of type `mps.cache.RedisCache` or `mps.cache.MATFileCache`.

Example: `c`

### **location** — Location name

'Remote' | 'Local'

Location of keys specified as an enumerated member of the class `mps.cache.Location`. The valid location options are either 'Remote' or 'Local'.

Example: 'Remote'

## Output Arguments

### **num** — Number of keys

integer

Total number of key-value pairs in the data cache or the number stored remotely or locally, returned as an integer.

## See Also

`bytes` | `get` | `isKey` | `keys` | `put`

### Topics

"Use a Data Cache to Persist Data" on page 7-2

### Introduced in R2018b

## countRemoteKeys

Count the number of keys stored on a remote persistence provider

### Syntax

```
count = countRemoteKeys(c)
```

### Description

`count = countRemoteKeys(c)` counts the number of keys stored on a remote persistence provider.

### Examples

#### Count the Number of Keys Stored on a Remote Persistence Provider

```
count = countRemoteKeys(c)
```

### Input Arguments

**c** — Data cache object  
`mps.cache.DataCache` object

Example:

### Output Arguments

`count` —

### See Also

**Introduced in R2018b**



## purge

Flush all local data to the persistence service

### Syntax

```
purgedKeys = purge(c)
```

### Description

`purgedKeys = purge(c)` flushes all local data to the persistence service and removes it locally.

### Examples

#### Flush All Local Data to the Persistence Service

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection','Redis','Port',4519);
start(ctrl)
c = mps.cache.connect('myCache','Connection','myRedisConnection');
```

Add keys and values to the cache.

```
put(c,'keyOne',10,'keyTwo',20,'keyThree',30,'keyFour',[400 500],'keyFive',magic(5))
```

Retain a few keys locally. For more information, see `retain`.

```
retain(c, {'keyOne','keyTwo'})
```

Modify the local keys and purge the data. Display the keys and values in the cache as a MATLAB table.

```
put(c,'keyOne',rand(3),'keyTwo',eye(10))
purgedKeys = purge(c)
tt = table(keys(c), get(c,keys(c)),'VariableNames',{'Keys','Values'})
display(c)
```

```
purgedKeys =
```

```
2×1 cell array
```

```
{'keyOne'}
{'keyTwo'}
```

```
tt =
```

```
5×2 table
```

```
Keys Values
```

```
'keyFive'    [ 5×5  double]
'keyFour'    [ 1×2  double]
'keyOne'     [ 3×3  double]
'keyThree'   [          30]
'keyTwo'     [10×10 double]
```

`c =`

RedisCache with properties:

```
Host: 'localhost'
Port: 4519
Name: 'myCache'
Operations: "read | write | create | update"
LocalKeys: {}
Connection: 'myRedisConnection'
```

Use `getp` instead of dot notation to access properties.

## Input Arguments

### **c** — Data cache

persistence provider specific data cache object

A data cache represented by a persistence provider specific data cache object.

Currently, Redis and MATLAB are the only supported persistence providers. Therefore, the cache objects will be of type `mps.cache.RedisCache` or `mps.cache.MATFileCache`.

Example: `c`

## Output Arguments

### **purgedKeys** — Purged keys

cell array of character vectors

List of keys that were written to the persistence service, returned as a cell array of character vectors.

## See Also

`clear` | `flush` | `keys` | `length` | `remove` | `retain`

### Topics

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

# put

Write key-value pairs to cache

## Syntax

```
put(c, key1, value1, ..., keyN, valueN)
put(c, keySet, valueSet)
```

## Description

`put(c, key1, value1, ..., keyN, valueN)` writes key-value pairs to cache. You can store any type of MATLAB data in a cache.

`put(c, keySet, valueSet)` writes key-value pairs to cache with keys from `keySet`, each mapped to a corresponding value from `valueSet`. The input arguments `keySet` and `valueSet` must have the same number of elements, with `keySet` having elements that are unique.

## Examples

### Write Series of Key-Value Pairs to Cache

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519);
start(ctrl)
c = mps.cache.connect('myCache', 'Connection', 'myRedisConnection');
```

Add keys and values to the cache and display them as a MATLAB table.

```
put(c, 'keyOne', 10, 'keyTwo', 20, 'keyThree', 30, 'keyFour', [400 500], 'keyFive', magic(5))
tt = table(keys(c), get(c, keys(c)), 'VariableNames', {'Keys', 'Values'})
```

```
tt =
```

```
5x2 table
```

Keys	Values
'keyFive'	[5x5 double]
'keyFour'	[1x2 double]
'keyOne'	[ 10]
'keyThree'	[ 30]
'keyTwo'	[ 20]

## Write Set of Keys and Corresponding Values to Cache

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection','Redis','Port',4519);
start(ctrl)
c = mps.cache.connect('myCache', 'Connection', 'myRedisConnection');
```

Add a set of keys and corresponding values to the cache and display them as a MATLAB table.

```
keySet = {'keyOne', 'keyTwo', 'keyThree', 'keyFour', 'keyFive'}
valueSet = {10, 20, 30, [400 500], magic(5)}
put(d, keySet, valueSet)
tt = table(keys(c), get(c, keys(c)), 'VariableNames', {'Keys', 'Values'})
```

```
tt =
```

```
5×2 table
```

Keys	Values
'keyFive'	[5×5 double]
'keyFour'	[1×2 double]
'keyOne'	[ 10]
'keyThree'	[ 30]
'keyTwo'	[ 20]

## Write Object to Cache

Create a class whose object you want to write to the Redis cache.

```
classdef BasicClass
    properties
        Value = pi;
    end
    methods
        function r = roundOff(obj)
            r = round([obj.Value],2);
        end
        function r = multiplyBy(obj,n)
            r = [obj.Value] * n;
        end
    end
end
```

Create an object of the class and assign a value to the Value property,

```
a = BasicClass
a.Value = 4
```

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection','Redis','Port',4519);
start(ctrl)
c = mps.cache.connect('myCache', 'Connection', 'myRedisConnection');
```

Add a key and the object that you created to the cache and retrieve the object.

```
put(c,'objKey',a)
objVal = get(c,'objKey')
```

```
objVal =
```

```
    BasicClass with properties:
```

```
    Value: 4
```

The output shows that there is no loss of information during writing an object to the cache and retrieving the object from the cache. The retrieved object contains the same information as the input object.

## Input Arguments

### **c** — Data cache

persistence provider specific data cache object

A data cache represented by a persistence provider specific data cache object.

Currently, Redis and MATLAB are the only supported persistence providers. Therefore, the cache objects will be of type `mps.cache.RedisCache` or `mps.cache.MATFileCache`.

Example: `c`

### **key** — Key

character vector

Key to add, specified as a character vector.

Example: `'keyFour'`

### **value** — Value

array

Value, specified as an array. `value` can be any valid MATLAB data type, including MATLAB objects.

Example: `[400, 500]`

### **keySet** — Keys

cell array of character vectors

Keys, specified as a cell array of character vectors.

Example: `{'keyOne', 'keyTwo', 'keyThree', 'keyFour', 'keyFive'}`

### **valueSet** — Values

cell array

Values, specified as comma-separated cell array. Each value may be any valid MATLAB data type, including MATLAB objects.

Example: `{10, 20, 30, [400 500], magic(5)}`

**See Also**

bytes | clear | get | keys | length | remove

**Topics**

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

## remove

Remove keys from cache

### Syntax

```
num = remove(c,keys)
```

### Description

`num = remove(c,keys)` removes keys and associated values from cache. There is no way to recover removed keys.

### Examples

#### Remove Keys from Cache

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection','Redis','Port',4519);
start(ctrl)
c = mps.cache.connect('myCache', 'Connection', 'myRedisConnection');
```

Add keys and values to the cache and display them as a MATLAB table.

```
put(c,'keyOne',10,'keyTwo',20,'keyThree',30,'keyFour',[400 500],'keyFive',magic(5))
tt = table(keys(c), get(c,keys(c)),'VariableNames',{'Keys','Values'})
```

```
tt =
```

```
5×2 table
```

Keys	Values
'keyFive'	[5×5 double]
'keyFour'	[1×2 double]
'keyOne'	[ 10]
'keyThree'	[ 30]
'keyTwo'	[ 20]

Remove two keys from cache `c` and display the remaining keys and values in the cache as a MATLAB table.

```
num = remove(c,{'keyThree','keyFour'})
tt = table(keys(c), get(c,keys(c)),'VariableNames',{'Keys','Values'})
```

```
num =
```

```
int64
```

```
2

tt =

3x2 table

  Keys          Values
-----
'keyFive'     [5x5 double]
'keyOne'       [          10]
'keyTwo'       [          20]
```

## Input Arguments

### **c** — Data cache

persistence provider specific data cache object

A data cache represented by a persistence provider specific data cache object.

Currently, Redis and MATLAB are the only supported persistence providers. Therefore, the cache objects will be of type `mps.cache.RedisCache` or `mps.cache.MATFileCache`.

Example: `c`

### **keys** — Keys to remove

cell array of character vectors

Keys to remove from cache, specified as a cell array of character vectors.

Example: `{ 'keyThree' , 'keyFour' }`

## Output Arguments

### **num** — Number of keys removed

integer

Number of keys removed, returned as an integer.

## See Also

`clear` | `get` | `keys` | `purge` | `put` | `retain`

### Topics

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**



# retain

Store remote keys from cache locally or return locally stored keys

## Syntax

```
retain(c, remoteKeys)
localKeys = retain(c)
```

## Description

`retain(c, remoteKeys)` stores keys from cache locally.

`localKeys = retain(c)` returns a cell array of keys stored locally.

## Examples

### Store Keys from Cache Locally and Check Local Keys

Start a persistence service that uses Redis as the persistence provider. The service requires a connection name and an open port. Once the service is running, you can connect to the service using the connection name and create a cache.

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519);
start(ctrl)
c = mps.cache.connect('myCache', 'Connection', 'myRedisConnection');
```

Add keys and values to the cache.

```
put(c, 'keyOne', 10, 'keyTwo', 20, 'keyThree', 30, 'keyFour', [400 500], 'keyFive', magic(5))
```

Retain a few keys locally and check local keys.

```
retain(c, {'keyThree', 'keyFour'})
localKeys = retain(c)
```

```
localKeys =
```

```
    1×2 cell array
```

```
    {'keyThree'}    {'keyFour'}
```

## Input Arguments

### **c** — Data cache

persistence provider specific data cache object

A data cache represented by a persistence provider specific data cache object.

Currently, Redis and MATLAB are the only supported persistence providers. Therefore, the cache objects will be of type `mps.cache.RedisCache` or `mps.cache.MATFileCache`.

Example: c

**remoteKeys — Keys**

cell array of character vectors

Remote keys to store locally, specified as a cell array of character vectors.

Example: { 'keyThree' , 'keyFour' }

**Output Arguments****localKeys — Keys**

cell array of character vectors

Locally stored keys, returned as a cell array of character vectors.

**Tips**

- As a performance optimization you may choose to temporarily store a set of keys and their values in your MATLAB session or worker instead of the persistence service. Keys *retained* in the this fashion will be automatically written to the persistence service (see `flush`) when MATLAB exits or when the first function call returns.
- Manually control the lifetime of retained keys with the `flush` and `purge` methods.

**See Also**

`clear` | `flush` | `purge` | `remove`

**Topics**

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

## mps.sync.mutex

Create a persistence service mutex

### Syntax

```
lk = mps.sync.mutex(mutexName, 'Connection', connectionName, Name, Value)
```

### Description

`lk = mps.sync.mutex(mutexName, 'Connection', connectionName, Name, Value)` creates a database advisory lock object.

### Examples

#### Create a Redis Mutex

First, create a persistence service controller object and use that object to start the persistence service.

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519);  
start(ctrl)
```

Use the connection name to create a persistence service mutex.

```
lk = mps.sync.mutex('myMutex', 'Connection', 'myRedisConnection')
```

```
lk =
```

```
TimedRedisMutex with properties:
```

```
Expiration: 10  
ConnectionName: 'myRedisConnection'  
MutexName: 'myMutex'
```

### Input Arguments

#### **mutexName** — Mutex name

character vector

Name of persistence service mutex, specified as a character vector.

Example: 'myMutex'

#### **connectionName** — Name of connection

character vector

Name of connection to persistence service, specified as a character vector.

Example: 'Connection', 'myRedisConnection'

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'Expiration', 10`

### Expiration — Time in seconds

positive integer

Expiration time in seconds after the lock is acquired.

Other clients will be able to acquire the lock even if you do not release it.

Example: `'Expiration', 10`

## Output Arguments

### `lk` — Mutex object

persistence service mutex object

A persistence service mutex object. If you use Redis as your persistence provider, `lk` will be a `mps.sync.TimedRedisMutex` object. If you use MATLAB as your persistence provider, `lk` will be a `mps.sync.TimedMATFileMutex` object.

## Tips

- A persistence service mutex allows multiple clients to take turns using a shared resource. Each cooperating client creates a mutex object with the same name using a connection to a shared persistence service. To gain exclusive access to the shared resource, a client attempts to acquire a lock on the mutex. When the client finishes operating on the shared resource, it releases the lock. To prevent lockouts should the locking client crash, all locks expire after a certain amount of time.
- Acquiring a lock on a mutex prevents other clients from acquiring a lock on that mutex but it does not lock the persistence service or any keys or values stored in the persistence service. These locks are advisory only and are meant to be used by cooperating clients intent of preventing data corruption. Rogue clients will be able to corrupt or delete data if they do not voluntarily respect the mutex locks.

## See Also

`acquire` | `mps.sync.TimedMATFileMutex` | `mps.sync.TimedRedisMutex` | `own` | `release`

## Topics

“Use a Data Cache to Persist Data” on page 7-2

## Introduced in R2018b

# mps.sync.TimedRedisMutex

Represent a Redis persistence service mutex

## Description

`mps.sync.TimedRedisMutex` is a synchronization primitive used to protect data in a Redis persistence service from being simultaneously accessed by multiple workers.

## Creation

Create a `mps.sync.TimedRedisMutex` object using `mps.sync.mutex`.

## Properties

### Expiration — Duration of lock in seconds

positive integer

This property is read-only.

Duration of advisory lock in seconds.

Example: 10

### ConnectionName — Name of connection

character vector

This property is read-only.

Name of connection to persistence service.

Example: 'myRedisConnection'

### MutexName — Name of mutex

character vector

This property is read-only.

Name of mutex, returned as a character vector.

Example: 'myMutex'

## Object Functions

<code>mps.sync.mutex</code>	Create a persistence service mutex
<code>acquire</code>	Acquire advisory lock on persistence service mutex
<code>own</code>	Check ownership of advisory lock on a persistence service mutex object
<code>release</code>	Release advisory lock on persistence service mutex

## Examples

### Create a Redis Lock Object

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519);  
start(ctrl)  
lk = mps.sync.mutex('myMutex', 'Connection', 'myRedisConnection')  
  
lk =
```

TimedRedisMutex with properties:

```
Expiration: 10  
ConnectionName: 'myRedisConnection'  
MutexName: 'myMutex'
```

### See Also

[acquire](#) | [mps.sync.TimedMATFileMutex](#) | [mps.sync.mutex](#) | [own](#) | [release](#)

### Topics

“Use a Data Cache to Persist Data” on page 7-2

### Introduced in R2018b

# mps.sync.TimedMATFileMutex

Represent a MAT-file persistence service mutex

## Description

`mps.sync.TimedMATFileMutex` is synchronization primitive used to protect data in a MAT-file database from being simultaneously accessed by multiple workers.

## Creation

Create a `mps.sync.TimedMATFileMutex` object using `mps.sync.mutex`.

## Properties

### Expiration — Duration of lock in seconds

positive integer

This property is read-only.

Duration of advisory lock in seconds.

Example: 10

### ConnectionName — Name of connection

character vector

This property is read-only.

Name of connection to persistence service.

Example: 'myRedisConnection'

### MutexName — Name of lock

character vector

This property is read-only.

Name of advisory lock, specified as a character vector.

Example: 'myMutex'

## Object Functions

<code>mps.sync.mutex</code>	Create a persistence service mutex
<code>acquire</code>	Acquire advisory lock on persistence service mutex
<code>own</code>	Check ownership of advisory lock on a persistence service mutex object
<code>release</code>	Release advisory lock on persistence service mutex

## Examples

### Create a MAT-File Lock Object

```
mctrl = mps.cache.control('myMATFileConnection','MatlabTest','Folder','c:\tmp')
start(mctrl)
lk = mps.sync.mutex('myMATFileMutex','Connection','myMATFileConnection')
```

```
lk =
```

```
TimedMATFileMutex with properties:
```

```
Expiration: 10
ConnectionName: 'myMATFileConnection'
MutexName: 'myMATFileMutex'
```

### See Also

`acquire` | `mps.sync.TimedRedisMutex` | `mps.sync.mutex` | `own` | `own` | `release` | `release`

### Topics

“Use a Data Cache to Persist Data” on page 7-2

### Introduced in R2018b



# acquire

Acquire advisory lock on persistence service mutex

## Syntax

```
TF = acquire(lk,timeout)
```

## Description

`TF = acquire(lk,timeout)` acquires an advisory lock and returns a logical 1 (`true`) if the lock was successful, and a logical 0 (`false`) otherwise. If the lock is unavailable, `acquire` will continue trying to acquire it for `timeout` seconds.

## Examples

### Apply Advisory Lock

First, create a persistence service controller object and use that object to start the persistence service.

```
ctrl = mps.cache.control('myRedisConnection','Redis','Port',4519);
start(ctrl)
```

Use the connection name to create a persistence service mutex.

```
lk = mps.sync.lock('myDbLock','Connection','myRedisConnection')
```

Try to acquire advisory lock. If lock is unavailable, retry acquiring for 20 seconds.

```
acquire(lk, 20);
```

```
TF =
```

```
    logical
```

```
    1
```

## Input Arguments

### lk — Mutex object

persistence service mutex object

A persistence service specific mutex object. If you use Redis as your persistence provider, `lk` will be a `mps.sync.TimedRedisMutex` object. If you use a MATLAB as your persistence provider, `lk` will be a `mps.sync.TimedMATFileMutex` object.

### timeout — Retry duration

positive integer

Duration after which to retry acquiring lock.

Example: 20

## **Output Arguments**

### **TF — Logical value**

logical array

TF has a logical 1 (`true`) if acquiring the advisory lock was successful, and a logical 0 (`false`) otherwise.

### **See Also**

`mps.sync.TimedMATFileMutex` | `mps.sync.TimedRedisMutex` | `mps.sync.mutex` | `own` | `release`

### **Topics**

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

## own

Check ownership of advisory lock on a persistence service mutex object

### Syntax

```
TF = own(lk)
```

### Description

`TF = own(lk)` returns a logical 1 (`true`) if you own an advisory lock on the persistence service mutex, and returns a logical 0 (`false`) otherwise.

### Examples

#### Check If You Own the Advisory Lock

First, create a persistence service controller object and use that object to start the persistence service.

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519);
start(ctrl)
```

Use the connection name to create a persistence service mutex.

```
lk = mps.sync.lock('myDbLock', 'Connection', 'myRedisConnection')
```

Check if you own the advisory lock.

```
TF = own(lk)
```

```
TF =
```

```
    logical
```

```
    0
```

### Input Arguments

#### lk — Mutex object

persistence service mutex object

A persistence service specific mutex object. If you use Redis as your persistence provider, `lk` will be a `mps.sync.TimedRedisMutex` object. If you use a MATLAB as your persistence provider, `lk` will be a `mps.sync.TimedMATFileMutex` object.

### Output Arguments

#### TF — Logical value

logical array

TF has a logical 1 (`true`) if you own the advisory lock on the persistence service mutex, and a logical 0 (`false`) otherwise.

### **See Also**

`acquire` | `mps.sync.TimedMATFileMutex` | `mps.sync.TimedRedisMutex` | `mps.sync.mutex` | `release`

### **Topics**

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

# release

Release advisory lock on persistence service mutex

## Syntax

```
TF = release(lk)
```

## Description

`TF = release(lk)` releases an advisory lock on a persistence service mutex. If the lock expires before you release it, `release` returns a logical 0 (`false`). If this occurs, it may indicate potential data corruption.

## Examples

### Release Advisory Lock

First, create a persistence service controller object and use that object to start the persistence service.

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519);
start(ctrl)
```

Use the connection name to create a persistence service mutex.

```
lk = mps.sync.lock('myDbLock', 'Connection', 'myRedisConnection')
```

Try to acquire advisory lock. If lock is unavailable, retry acquiring for 20 seconds.

```
acquire(lk, 20);
```

Release lock.

```
TF = release(lk)
```

```
TF =
```

```
    logical
```

```
     1
```

## Input Arguments

### lk — Mutex object

persistence service mutex object

A persistence service specific mutex object. If you use Redis as your persistence provider, `lk` will be a `mps.sync.TimedRedisMutex` object. If you use a MATLAB as your persistence provider, `lk` will be a `mps.sync.TimedMATFileMutex` object.

## **Output Arguments**

### **TF — Logical value**

logical array

TF has a logical 1 (`true`) if releasing the advisory lock was successful, and a logical 0 (`false`) otherwise.

### **See Also**

`acquire` | `mps.sync.TimedMATFileMutex` | `mps.sync.TimedRedisMutex` | `mps.sync.mutex` | `own`

### **Topics**

“Use a Data Cache to Persist Data” on page 7-2

**Introduced in R2018b**

# MATLAB Client

---

- “Connect MATLAB Session to MATLAB Production Server” on page 9-2
- “Execute Deployed MATLAB Functions” on page 9-5
- “Configure Client-Server Communication” on page 9-11
- “Application Access Control” on page 9-14
- “Execute Deployed Functions Using HTTPS” on page 9-17
- “Manage Add-Ons” on page 9-20
- “Deploy Add-Ons” on page 9-25

## Connect MATLAB Session to MATLAB Production Server

MATLAB Client for MATLAB Production Server makes the functions deployed on on-premises MATLAB Production Server instances available in your MATLAB session.

### When to Use MATLAB Client for MATLAB Production Server

MATLAB Client for MATLAB Production Server enables you to do the following:

- Scale with demand: Shift computationally intensive work from MATLAB desktop to server-class machines or scalable infrastructure.
- Centralize algorithm management: Install MATLAB functions that contain your algorithms on a central server and then run them from any MATLAB desktop, ensuring consistent usage and making upgrades easier.
- Protect intellectual property: Protect algorithms deployed to the server using encryption.

Using MATLAB Client for MATLAB Production Server is less suitable for algorithms that have the following characteristics:

- The algorithms are called several times from inside a loop.
- The algorithms require resources such as files or hardware that are available only on a single machine or to a single person.
- The algorithms rely on the MATLAB desktop or MATLAB graphics, or use data from a MATLAB session.

### Install MATLAB Client for MATLAB Production Server

Install the MATLAB Client for MATLAB Production Server support package from the MATLAB Add-On Explorer. For information about installing add-ons, see “Get and Manage Add-Ons” (MATLAB).

After your installation is complete, find examples in *support\_package\_root\toolbox\mps\matlabclient\demo*, where *support\_package\_root* is the root folder of support packages on your system. Access the documentation by entering the `doc` command at the MATLAB command prompt or by clicking the Help button. In the Help browser that opens, navigate to MATLAB Client for MATLAB Production Server under **Supplemental Software**.

### Connect MATLAB Session to MATLAB Production Server

MATLAB Client for MATLAB Production Server uses MATLAB add-ons to connect a MATLAB session to MATLAB functions deployed on server instances. The connection between a server instance and a MATLAB desktop session consists of two parts:

- 1 A MATLAB Production Server deployable archive that publishes one or more functions.
- 2 A MATLAB add-on that makes those functions available in MATLAB.

You must include a MATLAB function signature file when you create the deployable archive. For more information, see “MATLAB Function Signatures in JSON”. The server instance that hosts the deployable archive must have the discovery service enabled. For more information, see “Discovery Service”.



You must install a MATLAB Production Server add-on to connect a MATLAB desktop session to an archive deployed on a server instance. For example, for an archive `mathfun` deployed to a server instance running on `myhost.mycompany.com` at port 31415, you can install the corresponding add-on with a single command:

```
>> prodserver.addon.install('mathfun','myhost.mycompany.com',31415);
```

Then, you can call the functions in that archive from the MATLAB desktop, script, and function files. For example, if the deployed archive contains a function `mymagic` that takes an integer input and returns a magic square, you can call `mymagic` from the MATLAB command prompt.

```
>> mymagic(3)
```

For a detailed example, see “Execute Deployed MATLAB Functions” on page 9-5.

## System Requirements

MATLAB Client for MATLAB Production Server has the same system requirements as MATLAB. For more information, see System Requirements for MATLAB.

## Synchronous Function Execution

MATLAB programs are synchronous. Given a sequence of MATLAB function calls, MATLAB waits for each function to complete before calling the next one. Therefore, the MATLAB Production Server add-on functions are also synchronous. The add-ons use the MATLAB Production Server RESTful API for synchronous function execution. For more information about the RESTful API, see “Synchronous Execution”.

## Supported Data Types

MATLAB Client for MATLAB Production Server supports all data types that the MATLAB Production Server RESTful API supports, which are as follows:

- Numeric types: `double`, `single`, all integer types, complex numbers, `NaN`, `Inf` and `-Inf`.
- Character arrays
- Logical
- Cell arrays
- Structures
- String arrays
- Enumerations
- Datetime arrays

## See Also

`prodserver.addon.install`

## More About

- “Execute Deployed MATLAB Functions” on page 9-5

- “Get and Manage Add-Ons” (MATLAB)
- “Execute Deployed Functions Using HTTPS” on page 9-17

## Execute Deployed MATLAB Functions

### In this section...

“Install MATLAB Client for MATLAB Production Server” on page 9-5

“Deploy MATLAB Function on Server” on page 9-5

“Install MATLAB Production Server Add-On for the Deployable Archive” on page 9-6

“Manage Installed Add-On” on page 9-8

“Invoke Deployed MATLAB Function” on page 9-9

This example shows how to use MATLAB Client for MATLAB Production Server to invoke a MATLAB function deployed on an on-premises MATLAB Production Server instance.

MATLAB Client for MATLAB Production Server uses MATLAB Production Server add-ons to communicate between a MATLAB client and a server instance. A MATLAB Production Server add-on makes the functions in an archive deployed on MATLAB Production Server available in MATLAB. A deployed archive and its corresponding MATLAB Production Server add-on have the same name.

Installing the MATLAB Production Server add-on in your MATLAB desktop environment allows you to use the functions from a deployed archive in MATLAB. Installing a MATLAB Production Server add-on creates proxy functions of the deployed functions locally. The proxy functions manage communication between the deployed MATLAB functions and the clients that invoke the deployed functions. A proxy function and its corresponding deployed function have the same name. Since the proxy functions are MATLAB functions, you can call them from the MATLAB command prompt, other functions, or scripts. You can also compile the functions and scripts that contain the proxy functions. You can install MATLAB Production Server add-ons using the `prodserver.addon.install` function at the MATLAB command prompt or using the **MATLAB Production Server Add-On Explorer** app.

Calling the proxy MATLAB function sends an HTTP request across the network to an active MATLAB Production Server instance. The server instance calls the MATLAB function in the deployable archive and passes to it the inputs from the HTTP request. The return value of the deployed MATLAB function follows the same path over the network in reverse.

The following example describes how to install MATLAB Production Server add-ons and execute a deployed MATLAB function.

### Install MATLAB Client for MATLAB Production Server

Install the MATLAB Client for MATLAB Production Server support package to your MATLAB desktop environment using the MATLAB Add-On Explorer. For information about installing add-ons, see “Get and Manage Add-Ons” (MATLAB).

### Deploy MATLAB Function on Server

- 1 Write a MATLAB function `mymagic` that uses the `magic` function to create a magic square.

```
function m = mymagic(in)
    m = magic(in);
end
```

- 2 Package the function `mymagic` in an archive named `mathfun`. You must include a MATLAB function signature file when you create the archive. For information about creating the function signature file, see “MATLAB Function Signatures in JSON”.
- 3 Deploy the archive `mathfun` on a running MATLAB Production Server instance. The server instance must have the discovery service enabled. For information about enabling the discovery service, see “Discovery Service”. The server administrator typically deploys the archive and configures the server.

For information on how to create and deploy the archive, see “Create Deployable Archive for MATLAB Production Server” on page 2-2 and “Share Deployable Archive”.

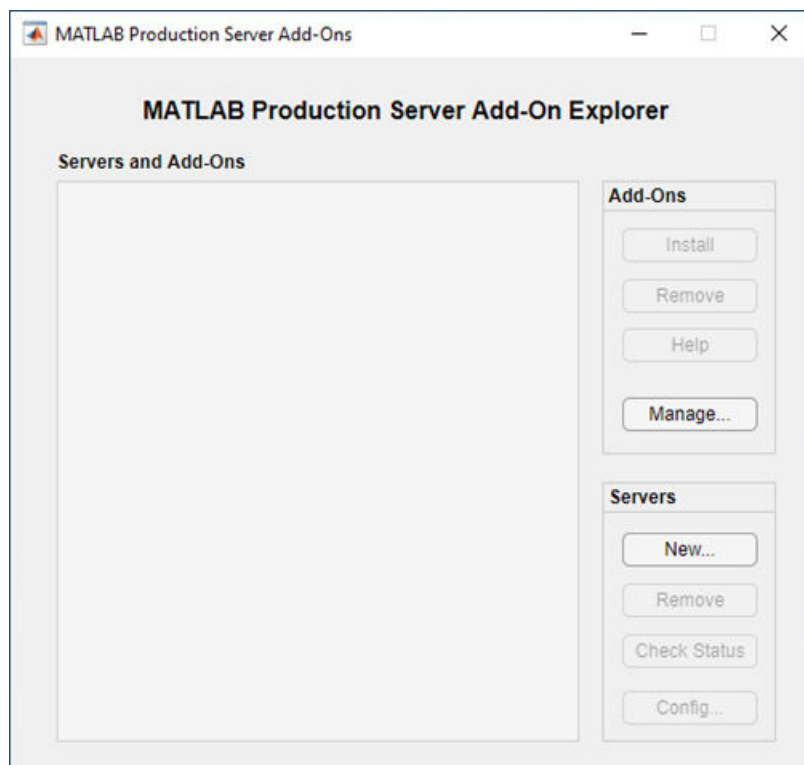
## Install MATLAB Production Server Add-On for the Deployable Archive

From your MATLAB desktop environment, install the MATLAB Production Server add-on for the deployed archive using the **MATLAB Production Server Add-On Explorer**. Installing the add-on makes the MATLAB functions deployed on the server available to your MATLAB client programs. The **MATLAB Production Server Add-On Explorer** is different from MATLAB Add-On Explorer.

### Launch MATLAB Production Server Add-On Explorer

From a MATLAB command prompt, launch the **MATLAB Production Server Add-On Explorer** using the command `prodserver.addon.Explorer`.

```
>> prodserver.addon.Explorer
```

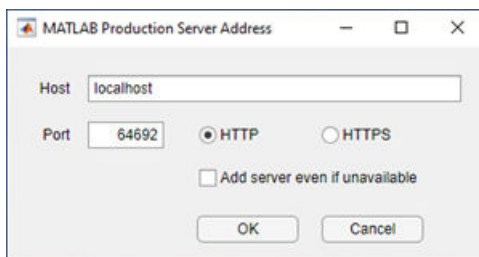


### Add Server Information

In the **MATLAB Production Server Add-On Explorer**, add information about the server that hosts the deployable archive `mathfun`.

- 1 In the **Servers** section, click **New**.
- 2 Enter the host name of the server in the **Host** box and the port number in the **Port** box. For example, for a server running on your local machine on port 64692, enter `localhost` for **Host** and 64692 for **Port**.
- 3 Click **OK** to add the server.
- 4 After you add the server, you can click **Check Status** to check the server status.

You can add multiple servers.

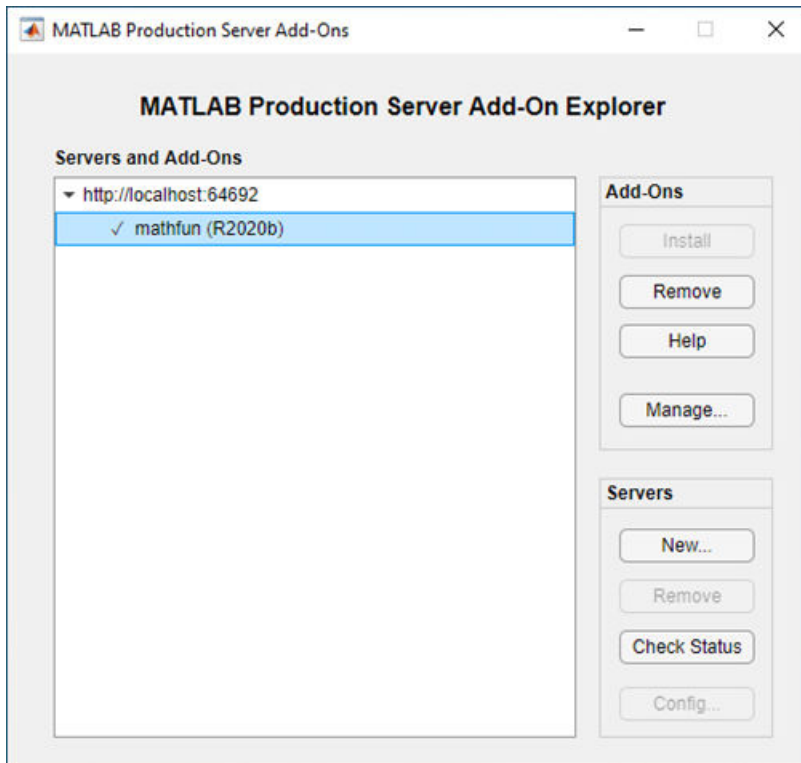


### Install Add-On

After you add a server, the **Servers and Add-Ons** section lists the server and the MATLAB Production Server add-ons that can communicate with the server. If you add multiple servers, this sections lists all the servers and the add-ons that can communicate with each server grouped under the server that hosts them.

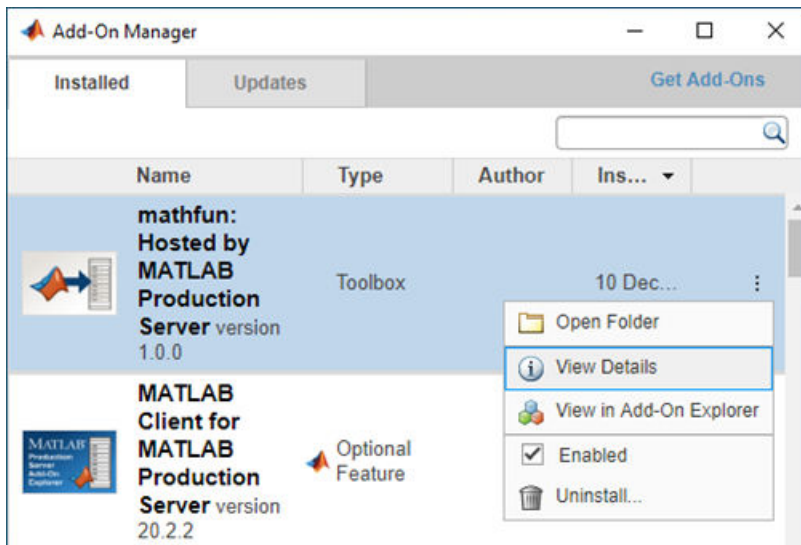
Install the `mathfun` add-on to make the MATLAB function `mymagic` from the deployable archive `mathfun` available in your MATLAB client programs.

- 1 Select the `mathfun` add-on.
- 2 In the **Add-Ons** section, click **Install**. This installs the add-on.



## Manage Installed Add-On

After you install a MATLAB Production Server add-on, the MATLAB Add-On Manager lists it. You can perform tasks such as enabling, disabling and uninstalling the add-on, and viewing details about the add-on. Viewing the add-on in Add-On Explorer is not supported.



## Invoke Deployed MATLAB Function

Installing an add-on creates proxy MATLAB functions locally that let you invoke MATLAB functions deployed on the server. You can call the proxy functions interactively from the MATLAB command prompt, other MATLAB functions, scripts, or standalone applications that in turn invoke the deployed MATLAB functions.

### Invoke Deployed MATLAB Function from Command Line

For example, to invoke the `mymagic` function hosted on the server, you can call the proxy `mymagic` function from the `matfun` add-on at the MATLAB command prompt.

```
>> mymagic(3)
```

This prints a 3 by 3 magic square.

### Invoke Deployed MATLAB Function from MATLAB Function

You can call the installed add-on proxy function in your MATLAB function and script. For example, write a simple MATLAB program `mytranspose.m` that creates a transpose of the magic square that you created using the proxy function `mymagic`.

```
function mytranspose
    A = mymagic(5);
    A.'
end
```

Running `mytranspose` prints the transpose of a 5 by 5 magic square.

```
>> mytranspose
```

### Invoke Deployed MATLAB Function from Standalone Executable

You can call the installed add-on proxy function in your MATLAB function and then create a standalone executable from the MATLAB function. For example, you can create a standalone executable from the `mytranspose` MATLAB client function using MATLAB Compiler.

```
>> mcc -m mytranspose
```

Run the standalone executable `mytranspose` at the system command prompt. You might need to install MATLAB Runtime if it is not installed on your machine. For more information, see [MATLAB Runtime](#).

```
C:\mytranspose> mytranspose
```

This prints a transpose of a 5 by 5 magic square.

You can configure the standalone executable to use time out values other than the default or use a different address for the server. For more information, see [“Configure Client-Server Communication”](#) on page 9-11.

You can find more examples in the `support_package_root\toolbox\mps\matlabclient\demo` folder, where `support_package_root` is the root folder of support packages on your system. You can access the documentation by entering the `doc` command at the MATLAB command prompt or clicking the Help button in MATLAB desktop. In the Help browser that opens, navigate to MATLAB Client for MATLAB Production Server under **Supplemental Software**.

## **See Also**

`prodserver.addon.Explorer` | `prodserver.addon.install`

## **More About**

- “Connect MATLAB Session to MATLAB Production Server” on page 9-2
- “Get and Manage Add-Ons” (MATLAB)
- “Discovery Service”
- “MATLAB Function Signatures in JSON”



## Configure Client-Server Communication

You can override the default configuration that MATLAB Production Server add-ons use for client-server communication by setting environment variables and updating the MATLAB Production Server add-on configuration file located on the client machine. You might want to override the default configuration if your network is reliable, if your application is time critical, or if you want to change the server information for add-ons packaged into standalone executables.

### Configure Timeouts and Retries

When you use MATLAB Client for MATLAB Production Server, the proxy functions in the MATLAB Production Server add-ons communicate with the functions of an archive deployed to a MATLAB Production Server instance. If the server takes too long to send a response, the client request times out. When a timeout occurs, the add-ons can report the error or silently try sending the request again.

MATLAB Production Server add-ons support two types of timeouts and one retry strategy. To override the default timeout durations and the default strategy for request retries, set MATLAB Production Server add-on environment variables.

#### Set Initial TCP Connection Timeout

Set the `PROSERVER_ADDON_CONNECT_TIMEOUT` environment variable to the number of seconds that an add-on function must wait before timing out when attempting to connect to a MATLAB Production Server instance. This is the initial TCP connection timeout.

By default, the operating system sets the TCP connection timeout value, typically, to 60 seconds or less, and might limit the value that you can set.

Typically, you do not need to set this value. If the server does not respond within the set time period, the add-ons generate an `MPS:MATLAB:AddOn:RequestTimeout` error.

#### Set Function Processing Timeout

Set the `PRODSERVER_ADDON_FUNCTION_TIMEOUT` environment variable to the number of seconds that an add-on function must wait for the deployed function to complete processing, which includes making the initial connection, and returning a response to the client.

The default behavior is to wait forever for the function to finish processing and never time out.

If your network is reliable or your application is time critical, you might set the environment variable so that the client request can time out earlier. Since the processing time for the add-on function includes the time to make a TCP connection with the server, do not set `PRODSERVER_ADDON_FUNCTION_TIMEOUT` to a non-zero value smaller than `PRODSERVER_ADDON_CONNECT_TIMEOUT`. If the deployed function does not return a complete response within the timeout value that you set, an `MPS:MATLAB:AddOn:RequestTimeout` error occurs.

#### Configure Function Retries

Set the `PRODSERVER_ADDON_FUNCTION_RETRIES` environment variable to the number of times that an add-on retries a single function call that times out. The add-on retries only those functions that time out. The add-on generates an error if a function fails for any other reason.

The default behavior specifies not to retry functions that time out and to report `MPS:MATLAB:AddOn:RequestTimeout` errors on the first timeout.

If the number of timeouts exceeds the value that you set, the add-on reports an `MPS:MATLAB:AddOn:RequestTimeout` error.

### Set Environment Variables

To control the timeouts in a MATLAB session, set the environment variables using the `setenv` function. For example:

Retry three times on timeout for the MATLAB function `mandelflake`:

```
>> setenv('PRODSERVER_ADDON_FUNCTION_RETRIES','3')
>> mandelflake
```

To control the timeouts in a standalone executable or software component, set the environment variables using commands specific to your operating system, typically `setenv` on Linux and macOS, and `set` on Windows. For example:

- Retry three times on timeout for the Linux standalone executable `mandelflake`:

```
% setenv PRODSERVER_ADDON_FUNCTION_RETRIES 3
% mandelflake
```

- Retry three times on timeout for a Windows standalone executable `mandelflake.exe`:

```
C:\> set PRODSERVER_ADDON_FUNCTION_RETRIES=3
C:\> mandelflake.exe
```

## Update Server Configuration

The MATLAB Production Server add-on configuration file specifies the association and dependency between the MATLAB Production Server add-on proxy functions and the MATLAB Production Server deployable archives from which you install the proxy functions. By default, the add-on proxy functions communicate with the MATLAB Production Server instance from which you install them. If the network address or the application access control configuration of the server instance changes, you can modify the configuration file to include the updated server information. For example, the network address of the server can change if you move from a testing environment to a production environment. The access control configuration can change if the Azure AD app registration credentials of the server change.

The configuration file lets you easily change server-specific information without rebuilding the deployable archive or reinstalling the add-on, since the mapping between an add-on and an archive is in the configuration file that is external to both. This external mapping is especially useful when you want to change the server information for add-on proxy functions that are packaged into a standalone executable or deployable software component, since standalone executables and deployable software components can also be shared and used on machines that are different from those that package them.

### Update Add-On Configuration File

The default name of the add-on configuration file is `prodserver_addon_config.json`. A sample configuration file follows.

```
{
  "Installed": {
```

```

    "Scheme": "http",
    "Host": "localhost",
    "Port": 9990,
    "Config": {
        "AccessTokenPolicy": "none",
        "ClientID": "",
        "IssuerURI": "",
        "ServerID": ""
    },
    "AddOns": {
        "name": "fractal (R2020b)",
        "uuid": "e33251o6-4297-47d2-9ec8-9df64195fce3",
        "archiveID": "fractal_311a3f55107d8d603cc3d91707bf2feb"
    }
},
"SchemaVersion": 1.2
}

```

The sample configuration file describes a single add-on `fractal` that requires MATLAB Runtime version R2020b and a deployable archive `fractal` hosted by a MATLAB Production Server instance at network address `localhost:9990`.

To update the network address of the server, update the values corresponding to the `Host` and `Port` fields. To update the access control configuration of the server, update the values in the `Config` object. If you do not manage the server, you can obtain these values from the server administrator. For more information about configuring access control when using the add-ons, see “Application Access Control” on page 9-14.

### Update Add-On Configuration File Location

The default location of the add-on configuration file is in the MATLAB user preference directory of the machine on which the add-on function is installed. To locate the preferences directory on your machine, run `prefdir` at the MATLAB command prompt.

You can save the add-on configuration file in a different location and also change the name of the add-on configuration file. To specify a different location or name than the default, set the `PRODSERVER_ADDON_CONFIG` environment variable. When setting the variable, you must specify the full path to the file from the root of the file system. You might save the add-on configuration file in a different location when you want to update the server configuration for add-on proxy functions that are packaged into standalone executables or shared components.

## See Also

### More About

- “Execute Deployed MATLAB Functions” on page 9-5
- “Execute Deployed Functions Using HTTPS” on page 9-17
- “Application Access Control” on page 9-14
- “Manage Add-Ons” on page 9-20

## Application Access Control

MATLAB Production Server uses Azure Active Directory (Azure AD) to restrict access to deployed applications to only certain groups of users. If access control is enabled on the server that a MATLAB client application communicates with, the client application must send a bearer token when it sends requests to the server. The bearer token identifies the user that is executing the client application. Based on the bearer token, the server grants or denies access to client applications for executing deployed applications.

### Prerequisites

- 1 Access control is enabled on the server. For more information, see “Application Access Control”.
- 2 The MATLAB Production Server add-on of the deployed application is installed on the client machine. For more information about installing add-ons, see “Execute Deployed MATLAB Functions” on page 9-5.

### Configure Access Control

Configure access control on the client machine to send a bearer token in server requests. You can send either a system-generated bearer token or specify a bearer token.

#### Use System-generated Bearer Token

To enable a client application that you write using MATLAB Client for MATLAB Production Server to send a system-generated bearer token to a server, you must set the Azure AD app registration credentials and set an access token policy. Obtain the `ServerID`, `ClientID` and `IssuerURI` of the Azure AD apps that your organization uses for user authorization from the MATLAB Production Server administrator or the Azure AD administrator of your organization. Typically, you must set these credentials once for each server instance that your MATLAB client applications communicate with.

- `ServerID` — Application ID of the server app registered on Azure AD that is used for user authorization. The `ServerID` value must be the same as the `appID` value in the access control configuration file present on the MATLAB Production Server instance. For more information, see “Access Control Configuration File”.
- `ClientID` — Application ID of the client app registered on Azure AD that is used for user authorization.
- `IssuerURI` — URI followed by the Azure AD tenant ID that the client uses to generate a bearer token for a user.

Run the `prodserver.addon.accessTokenPolicy` function at the MATLAB command prompt to set the Azure AD app registration credentials and specify the `automatic` access token policy to use a system-generated bearer token. Also specify as arguments, the host name and port of the MATLAB Production Server instance that your add-on communicates with.

```
>> prodserver.addon.accessTokenPolicy('localhost',51133,'automatic',...
'ClientID','0d963963-e439-41d0-822c-b15ayu8937c3',...
'ServerID','d19d8po0-7977-4213-a05a-10kjna82fbaf',...
'IssuerURI','https://login.microsoftonline.com/yourcompany.com')
```

The **MATLAB Production Server Add-On Explorer** does not support setting the Azure AD app registration credentials.

## Specify Bearer Token

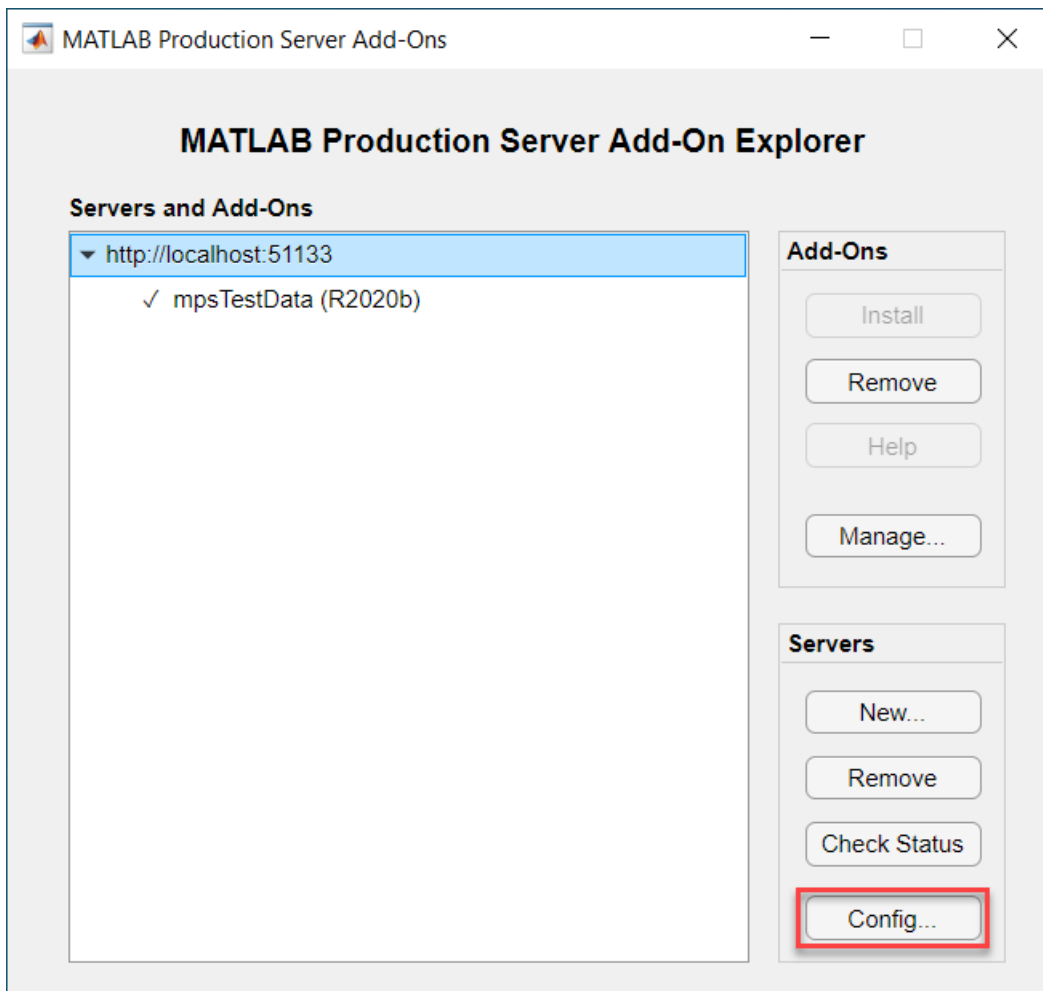
If you want to specify your own bearer token, you can use the `prodserver.addon.accessTokenPolicy` function to do so.

```
>> prodserver.addon.accessTokenPolicy('localhost',51133,'your_access_token')
```

## Set Access Token Policy Using MATLAB Production Server Add-On Explorer App

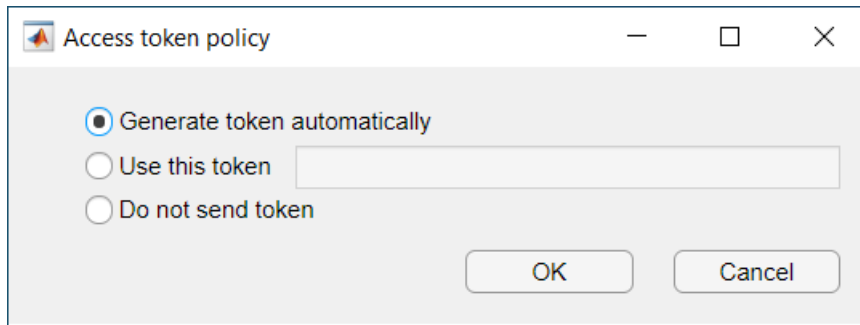
You can use the **MATLAB Production Server Add-On Explorer** to switch between using a system-generated bearer token or specifying your own bear token.

- 1 In the **MATLAB Production Server Add-On Explorer** app, select the server that you want your client applications to communicate with, then click **Config**.



- 2 In the dialog box that opens, configure the access token policy. Choose **Generate token automatically** to let the software generate an access token for you, or choose **Use this token** and specify the access token. Click **OK** to save your selection.

You must set the `serverID`, `clientID`, and `IssuerURI` parameters from the command line before making a selection to use the system-generated token.



### See Also

`prodserver.addon.accessTokenPolicy`

### More About

- “Application Access Control”
- “Execute Deployed MATLAB Functions” on page 9-5

## Execute Deployed Functions Using HTTPS

Connecting to a MATLAB Production Server instance over HTTPS provides a secure channel for executing MATLAB functions. To establish an HTTPS connection with a MATLAB Production Server instance:

- 1 Ensure that the server instance is configured to use HTTPS. For more information, see “Enable HTTPS”.
- 2 If the server instance uses a self-signed SSL certificate or if the root certificate of the server is not present in the trust store of the client machine, you must save the server certificate on the client machine.
- 3 Install MATLAB Production Server add-ons using HTTPS.

MATLAB Client for MATLAB Production Server does not support sending a client certificate to the server. Therefore, you cannot use MATLAB Client for MATLAB Production Server to install add-ons from a server or execute functions deployed to a server that has client authentication enabled.

### Save SSL Certificate of Server

Before your client application can send HTTPS requests to a server instance, the root SSL certificate of the server must be present in the Windows Trusted Root Certification Authorities certificate store or Linux trust store of the client machine. If the server uses a self-signed SSL certificate or if the root certificate of the server signed by a certificate authority (CA) is not present in the Windows certificate store, obtain the server certificate from the MATLAB Production Server administrator or export the certificate using a browser, then add it to the certificate store or the trust store.

#### Export and Save SSL Certificate

You can use any browser to save the server certificate on the client machine. The procedure to save the certificate using Google Chrome™ follows.

- 1 Navigate to the server instance URL `https://your_server_FQDN:port/api/health` using Google Chrome.
- 2 In the Google Chrome address bar, click the padlock icon or the warning icon, depending on whether the server instance uses a CA-signed SSL certificate or a self-signed SSL certificate.
- 3 Click **Certificate > Details > Copy to File**. Doing so opens a wizard that lets you export the SSL certificate. Click **Next**.
- 4 Select the format to export the certificate and click **Next**.
- 5 Specify the location and file name to export the certificate, then click **Next**.
- 6 Click **Finish** to complete exporting the certificate.

#### Add Certificate to Windows Certificate Store

You can use a certificate management tool or Microsoft Management Console (MMC) to add the server certificate to the Windows certificate store. The procedure to add the certificate using MMC follows.

- 1 Open MMC from your Windows machine.
- 2 Click **File > Add/Remove Snap-in**. Doing so opens the **Add or Remove Snap-ins** window.
- 3 In the **Add or Remove Snap-ins** window:

- a Click **Certificates** from the left pane, then click **Add**.
  - b Select **Computer account**, then click **Finish**. Doing so adds **Certificates(Local Computer)** to the right pane.
  - c Click **OK**. Doing so takes you to the home window.
- 4 In the left pane of the home window, under **Console Root**, double click **Certificates(Local Computer)**. Doing so opens all the certificate folders located in the local machine.
  - 5 Select **Trusted Root Certification Authorities > More Actions > All Tasks > Import**. Doing so opens the Certificate Import Wizard.
  - 6 Click **Next**, then select the location of your server certificate.
  - 7 Click **Next** to import the certificate in the Trusted Root Certification Authorities certificate store.

### Specify Custom Path to Certificate

MATLAB Client for MATLAB Production Server searches the default trust store for the server certificate, but also supports specifying the full path to a certificate file. Typically, you want to specify a path to the server certificate during testing. To do so, set the `CertificateFile` property using the `prodserver.addon.set` function. The value of the `CertificateFile` property persists between MATLAB sessions.

```
>> prodserver.addon.set('CertificateFile', '/path/to/my/certificate.pem')
```

## Install Add-On Using HTTPS

The default protocol for communication with the server is HTTP. If the server uses HTTPS, you must install an add-on from that server using HTTPS. Your MATLAB session can use HTTP with one server and HTTPS with another server simultaneously.

### Install Add-On Using Command Line

Use the `prodserver.addon.install` function and set the `TransportLayerSecurity` property to `true` to use HTTPS.

```
>> prodserver.addon.install('fractal', 'localhost', 9920, 'TransportLayerSecurity', true)
```

### Install Add-On Using Graphical Interface

- 1 In the **Servers** section in the **MATLAB Production Server Add-On Explorer** app, click **New**. Doing so opens a dialog box where you enter details about the MATLAB Production Server instance that the MATLAB client wants to communicate with using HTTPS.
- 2 Enter the host name and port number of the server instance, select **HTTPS**, then click **OK**. Doing so enables HTTPS communication between your MATLAB client and the server instance.

## Manage Default Protocol for Client-Server Communication

Set the protocol for client-server communication to HTTPS by using the `prodserver.addon.set` function and setting the `TransportLayerSecurity` property to `true`. The protocol setting persists between MATLAB sessions.

```
prodserver.addon.set('TransportLayerSecurity', true);
```



View the current value of the `TransportLayerSecurity` property using the `prodserver.addon.get` function.

```
prodserver.addon.get('TransportLayerSecurity');
```

```
ans =
```

```
    logical  
         1
```

## See Also

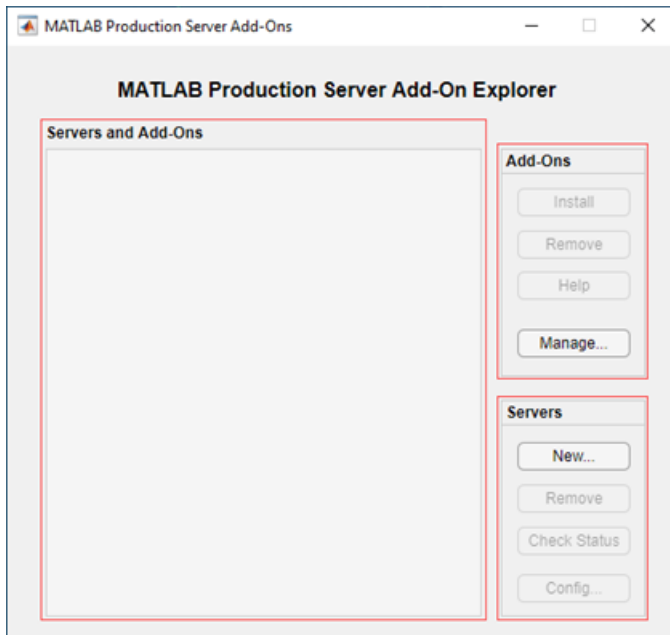
`prodserver.addon.accessTokenPolicy` | `prodserver.addon.get` |  
`prodserver.addon.install` | `prodserver.addon.set`

## External Websites

- “Execute Deployed MATLAB Functions” on page 9-5
- “Application Access Control” on page 9-14

## Manage Add-Ons

The **MATLAB Production Server Add-On Explorer** provides a graphical interface to find, install, and manage MATLAB Production Server add-ons. It requires MATLAB Client for MATLAB Production Server. To open **MATLAB Production Server Add-On Explorer**, enter `proserver.addon.Explorer` at the MATLAB command prompt.



- The **Servers** section lets you add and remove MATLAB Production Server instances from which you can install add-ons, check server status, and configure access control for executing deployed applications.
- The **Add-Ons** section provides options to install and remove MATLAB Production Server add-ons, view help text for the add-ons, and manage add-ons using the **MATLAB Add-On Manager**.
- The **Servers and Add-Ons** section lists the add-ons grouped by server.

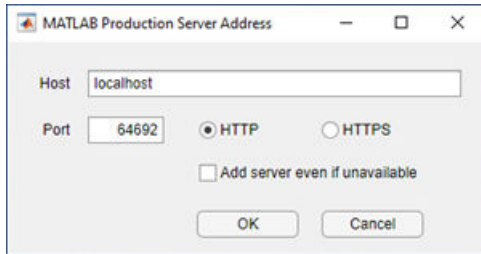
## Install Add-Ons

Installing MATLAB Production Server add-ons in your MATLAB desktop environment allows you to use the functions from an archive deployed to a MATLAB Production Server instance in MATLAB. You must add information about the server instances before you can install add-ons from them.

### Add Server

- 1 In the **Servers** section of **MATLAB Production Server Add-on Explorer**, click **New**.
- 2 Enter the host name of the server in the **Host** box. Use a name such as `localhost` or `addons.yourcompany.com`, or a numeric address such as `127.0.0.1`.
- 3 Enter the port number in the **Port** box. Port numbers are integers between 1 and 65535.
- 4 Select the protocol, HTTP or HTTPS, that the server uses. You can find which protocol a server expects by examining the MATLAB Production Server configuration file `main_config` or by making a request to the GET Discovery Information from a browser.

- 5 Select **Add server even if unavailable**, only if you want to add a server that is not yet available. You might do this if you plan to start the server later.
- 6 Click **OK** to add the server.



To check the server status, select the server from the **Servers and Add-Ons** section, then click **Check Status**.

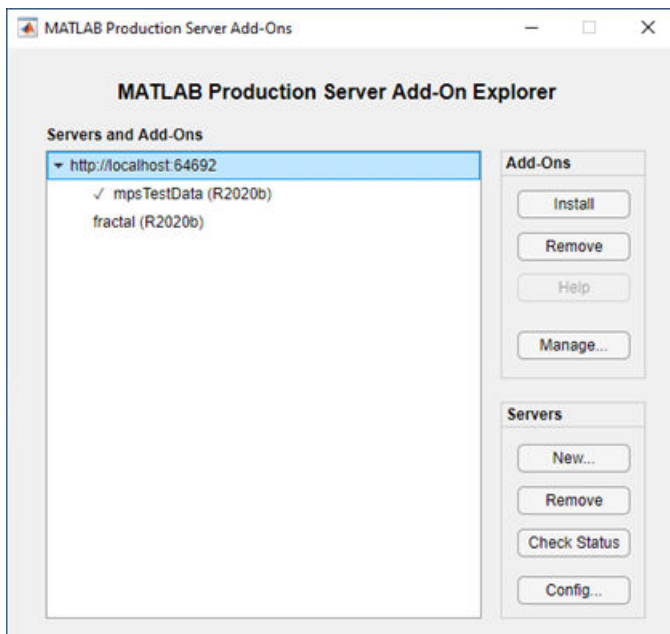
To remove a server, select the server from the **Servers and Add-Ons** section, then click **Remove**. Removing a server also removes the add-ons installed from the server.

### Install Add-On

After you add a server, the **Servers and Add-Ons** section lists the server and the MATLAB Production Server add-ons that can communicate with the server. If you add multiple servers, this section lists all the servers and the add-ons that can communicate with each server, grouped under the server that hosts them.

To install an add-on, select the add-on from the **Servers and Add-Ons** section, then click **Install** in the **Add-Ons** section.

The following graphic shows a server instance running at `http://localhost:64692` that has the `mpsTestdata` and the `fractal` add-ons available. The check mark indicates that the `mpsTestdata` add-on is installed on the client machine.



For information about installing add-ons from the MATLAB command prompt, see `prodserver.addon.install`.

For information about executing deployed applications using the installed add-ons, see “Execute Deployed MATLAB Functions” on page 9-5 and “Execute Deployed Functions Using HTTPS” on page 9-17.

## Remove Add-Ons

To remove add-ons, select them from the **Servers and Add-Ons** section, then click **Remove** from the **Add-Ons** section. Functions from removed add-ons are no longer available to MATLAB.

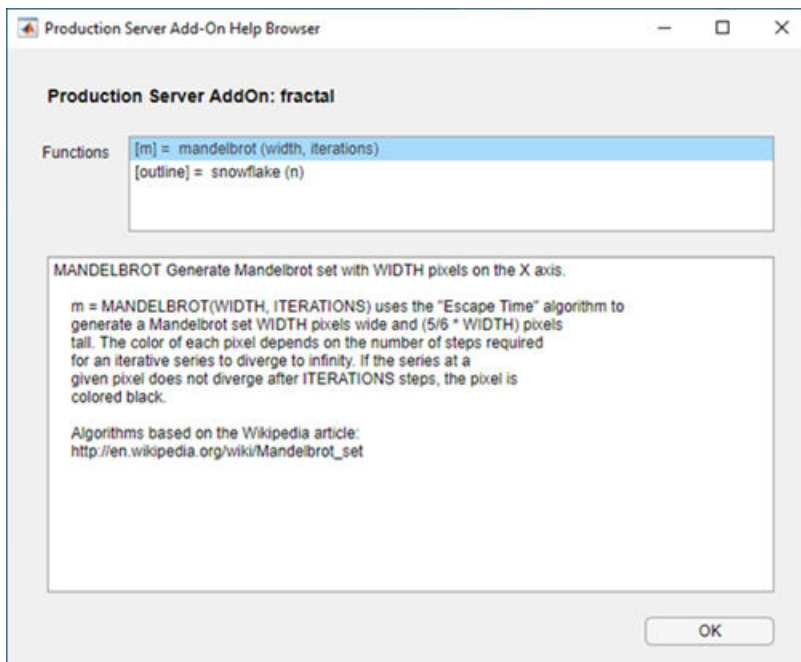
For information about uninstalling add-ons from the MATLAB command prompt, see `prodserver.addon.uninstall`.

## Get Information about Add-Ons

To view information about an add-on, select the add-on from the **Servers and Add-Ons** section, then click **Help** from the **Add-Ons** section. Add-Ons do not need to be installed for you to browse their help.

Select a function to view the help text written by the function author.

The following graphic shows the help text for the `mandelbrot` function present in the `fractal` add-on.



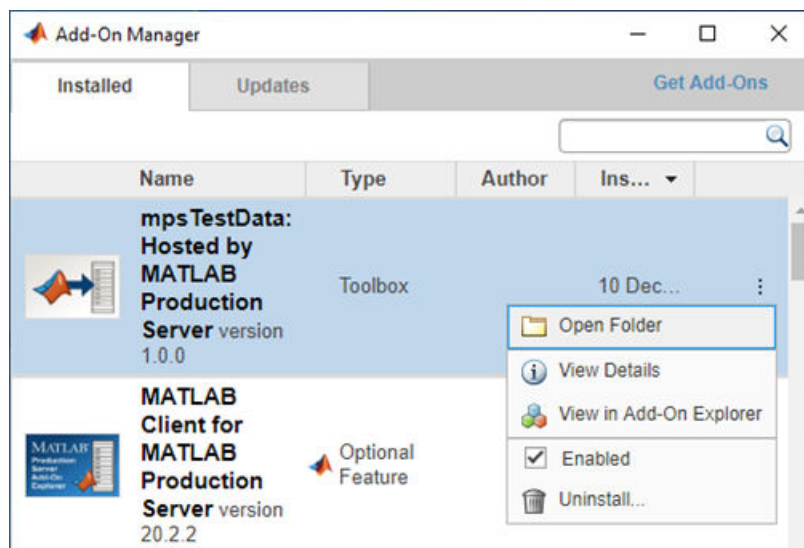
## Manage Add-Ons

After you install a MATLAB Production Server add-on, the **MATLAB Add-On Manager** lists it. You can perform tasks such as enabling, disabling and uninstalling the add-on, and viewing add-on details.

Removing an add-on from the **MATLAB Production Server Add-On Explorer** is equivalent to uninstalling from the **MATLAB Add-On Manager**.

Disabling an add-on removes the add-on from the MATLAB path.

The following graphic shows the `mpsTestData` add-on in the **MATLAB Add-On Manager**.



## Manage Access to Applications Deployed on Server

If a client program that you write using MATLAB Client for MATLAB Production Server wants to execute applications deployed to a server that has application access control enabled, the client must send a bearer token in server requests. The bearer token identifies the client. To specify a bearer token, select the server from the **Servers and Add-Ons** section, then click **Config** in the **Servers** section. For more information on how to specify bearer tokens, see “Application Access Control” on page 9-14.

For information about specifying a bearer token from the MATLAB command prompt, see `prodserver.addon.accessTokenPolicy`.

### See Also

`prodserver.addon.Explorer` | `prodserver.addon.accessTokenPolicy` |  
`prodserver.addon.availableAddOns` | `prodserver.addon.install` |  
`prodserver.addon.isInstalled` | `prodserver.addon.uninstall`

### More About

- “Execute Deployed MATLAB Functions” on page 9-5

- “Execute Deployed Functions Using HTTPS” on page 9-17
- “Application Access Control” on page 9-14

## Deploy Add-Ons

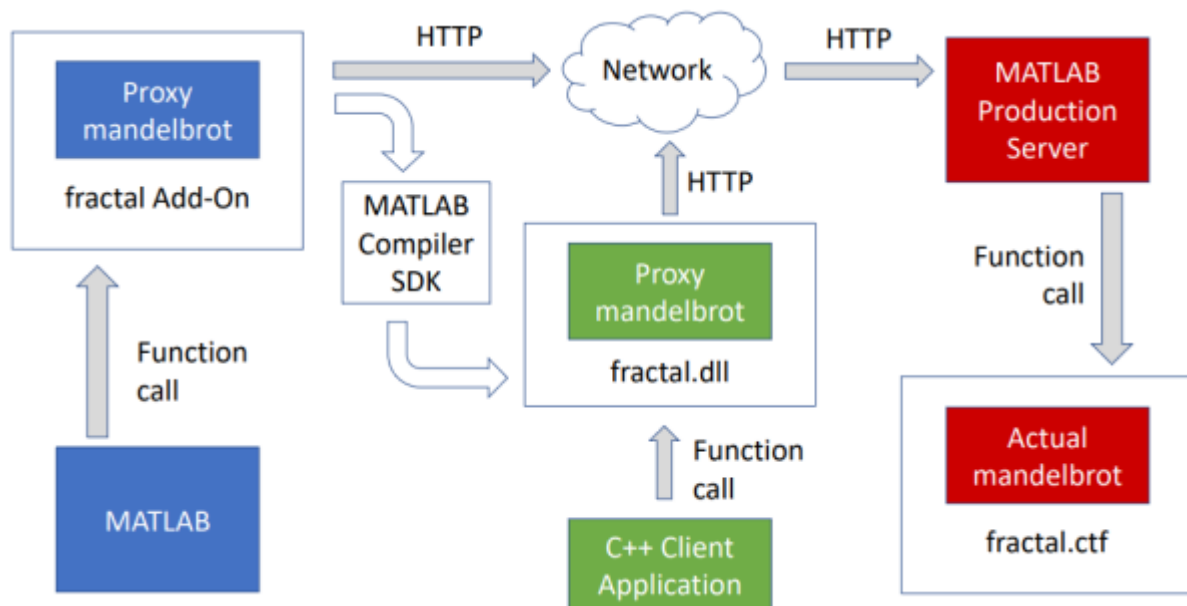
MATLAB deployment tools such as MATLAB Compiler and MATLAB Compiler SDK package MATLAB functions for deployment to environments external to the MATLAB desktop. These deployment tools can also package the proxy functions that MATLAB Production Server add-ons install to create deployable software components that require both the external environment and an active MATLAB Production Server instance.

For example, consider a deployable archive `fractal.ctf` that contains a MATLAB function `mandelbrot` hosted on a MATLAB Production Server instance.

You can install the `fractal` add-on on a client machine from the `fractal` archive using MATLAB Client for MATLAB Production Server. Installing the `fractal` add-on installs the proxy `mandelbrot` function on your machine. Then, you can write a client program in MATLAB that uses the proxy `mandelbrot` function.

You can also package the proxy `mandelbrot` function into a shared library, for example, `fractal.dll`, using MATLAB Compiler SDK. Then, you can write a C++ client program that uses `fractal.dll`.

The following diagram shows the MATLAB client (in blue) and the C++ client (in green) calling the same proxy `mandelbrot` function to communicate with the `mandelbrot` function deployed to a MATLAB Production Server instance.



The following examples show how to package installed proxy functions into a standalone executable, a shared library, and a deployable archive. The examples use files in the `support_package_root\toolbox\mps\matlabclient\demo` folder on your system. The `demo` folder contains the following folders:

- `fractal` — Contains `mandelbrot` and `snowflake` MATLAB functions. The `mandelbrot` function generates a Mandelbrot set and the `snowflake` function generates the outline of a Koch snowflake. You package these MATLAB functions into a MATLAB Production Server deployable archive.

- `mandelflake` — Contains the `mandelflake` MATLAB function that displays the Mandelbrot set and the Koch snowflake. You package the `mandelflake` function into a standalone executable.
- `fractalViewer` — Contains the `twoFractals` MATLAB function that displays the Mandelbrot set and the Koch snowflake based on input arguments that you specify. You package the `twoFractals` function into a shared library and a deployable archive.

## Prerequisites

The examples require that you have the `fractal` MATLAB Production Server add-on available in your MATLAB session. The examples package the proxy functions from the `fractal` add-on into a standalone executable, a shared library, and a deployable archive. To make the `fractal` add-on available in MATLAB:

- 1 Package the `mandelbrot` and `snowflake` MATLAB functions from the `\demo\fractal\` folder into a deployable archive called `fractal` using the **Production Server Compiler** app. You must include a MATLAB function signature file when you create the archive. For more information about packaging archives, see “Package Deployable Archives with Production Server Compiler App” on page 2-5.
- 2 Deploy the `fractal` archive to a MATLAB Production Server instance. For more information about deploying the archive, see “Share Deployable Archive”.

Confirm with the server administrator that the discovery service is enabled on the server. For more information, see “Discovery Service”.

- 3 Install the `fractal` add-on in your MATLAB desktop. For more information about installing add-ons, see `prodserver.addon.install`. For a detailed example about installing MATLAB Production Server add-ons, see “Execute Deployed MATLAB Functions” on page 9-5.

You can verify that the `fractal` add-on is available in your MATLAB session by running `prodserver.addon.availableAddOns`. To test your installation of the `fractal` add-on, you can run the example MATLAB function `mandelflake` that is in `\demo\mandelflake` at the MATLAB command prompt.

The standalone executable and shared library require MATLAB Runtime. Install MATLAB Runtime on your machine if you have not already done so. For more information, see MATLAB Runtime.

## Create Standalone Executables That Use Add-Ons

This example shows how to package a proxy function that a MATLAB Production Server add-on installs, into a standalone executable to invoke a MATLAB function hosted on a MATLAB Production Server instance. This example requires MATLAB Compiler. You can run standalone executables on computers that do not have MATLAB installed.

- 1 For this example, copy the contents of the `support_package_root\toolbox\mps\matlabclient\demo\mandelflake` folder to a separate writeable location on your system, for example, to a folder called `mandelflake`.
- 2 Navigate to the writeable `mandelflake` folder from the MATLAB command prompt. The `mandelflake` folder contains a MATLAB function called `mandelflake`. Use the `mcc` command to create a standalone executable called `mandelflake` from the `mandelflake` MATLAB function.

```
>> cd mandelflake
>> mcc -m mandelflake
```



This command produces an executable file `mandelflake.exe` on a Windows system.

On Linux and Mac OS, it produces an executable called `mandelflake`.

- 3 Run the executable at the system command prompt to display the Mandelbrot set and Koch snowflake.

```
C:\mandelflake> mandelflake
```

Two windows appear, one containing the Mandelbrot set and one displaying the Koch snowflake.

## Create Shared Libraries or Software Components That Use Add-Ons

This example shows how to package a proxy function that a MATLAB Production Server add-on installs, into a shared library, then use the shared library in a C++ client to invoke a MATLAB function hosted on a MATLAB Production Server instance. This example requires MATLAB Compiler SDK and a supported C++ compiler. For a list of supported C++ compilers, see [Supported and Compatible Compilers](#). MATLAB Compiler SDK creates software components, such as shared libraries, from MATLAB functions.

- 1 For this example, copy the contents of the `support_package_root\toolbox\mps\matlabclient\demo\fractalViewer` folder to a separate writeable location on your system, for example, to a folder called `fractalViewer`. The `fractalViewer` folder contains the following:
  - A MATLAB function `twoFractals` that displays images of the Mandelbrot set and the Koch snowflake based on the input arguments to the function
  - A C++ application `fractalViewer` that invokes the `twoFractals` function with the required input arguments
- 2 Navigate to the writeable `fractalViewer` folder from the MATLAB command prompt. Use the `mcc` command to create a shared library called `twoFractals.lib` from the `twoFractals.m` MATLAB function.

```
>> cd fractalViewer
>> mcc -W cpplib:twoFractals twoFractals.m
```

- 3 The `twoFractals` shared library requires a client to utilize its public interface. Use the `mbuild` function to compile and link the `fractalViewer` C++ application against the `twoFractals` shared library. The `fractalViewer` C++ application invokes the `twoFractals` function with the appropriate inputs.

```
>> mbuild fractalViewer.cpp twoFractals.lib
```

This command produces an executable file `fractalViewer.exe` and a shared library `twoFractals.dll` on a Windows system.

On Linux, it produces an executable `twoFractals.so` and a shared library `fractalViewer`. On Mac OS, it produces an executable `twoFractals.dylib` and a shared library `fractalViewer`.

- 4 Run the `fractalViewer` executable at the system command prompt to display the Mandelbrot set and Koch snowflake.

```
C:\fractalViewer> fractalViewer
```

Two windows appear, one containing the Mandelbrot set and one displaying the Koch snowflake.

## Create Deployable Archives That Use Add-Ons

This example shows how to package a proxy function that is in one MATLAB Production Server add-on into a MATLAB Production Server archive, from which you can install a second MATLAB Production Server add-on. In this case, the proxy functions of the second add-on call the proxy functions of the first add-on, which in turn call the actual functions (functions hosted on the first MATLAB Production Server instance) of the first add-on. With this feature, you can chain together multiple MATLAB Production Server archives. However, longer chains require more network resources and run more slowly. Application access control is not supported for deployed archives that contain the add-on proxy functions.

This example requires MATLAB Compiler SDK.

- 1 For this example, copy the contents of the `support_package_root\toolbox\mps\matlabclient\demo\fractalViewer` folder to a separate writeable location on your system, for example, to a folder called `fractalViewer`. The `fractalViewer` folder contains a MATLAB function `twoFractals` that displays images of the Mandelbrot set and the Koch snowflake.
- 2 Create a MATLAB function signature file `twoFractalsFunctionSignatures.json` in the writeable `fractalViewer` folder. You require a MATLAB function signature file when you create a deployable archive of the `twoFractals` function. For more information, see “MATLAB Function Signatures in JSON”. A sample MATLAB function signature file follows.

### `twoFractalsFunctionSignatures.json`

```
// Function Signatures
// To optionally specify argument types and/or sizes, search for "type"
// and insert the appropriate specifiers inside the brackets. For example:
//
//   "type": ["double", "size=1,1"]
//
// To modify function or parameter help text, search for "purpose" and edit
// the values.
//
// JSON-formatted text below this line.
{
  "_schemaVersion": "1.1.0",
  "twoFractals": {
    "inputs": [
      {
        "name": "maxIterations",
        "type": [],
        "purpose": ""
      },
      {
        "name": "width",
        "type": [],
        "purpose": ""
      },
      {
        "name": "complexity",
        "type": [],
        "purpose": ""
      }
    ],
    "outputs": [],
    "purpose": " TWOFRACTALS Display Mandelbrot set and Koch snowflake.\n"
```

```
    }  
}
```

- 3 Navigate to the writeable `fractalViewer` folder from the MATLAB command prompt. Use the `mcc` command to create a deployable archive `twoFractal.ctf` from the `twoFractals.m` MATLAB function.

```
>> cd fractalViewer
```

```
>> mcc('-W','CTF:twoFractals,DISCOVERY:twoFractalsFunctionSignatures.json','-U','twoFractals.m')
```

- 4 Copy the resulting archive, `twoFractals.ctf`, to the `auto_deploy` folder of a MATLAB Production Server instance.
- 5 Then, install the `twoFractals` MATLAB Production Server add-on. For example, if your MATLAB Production Server instance has the network address `localhost:9910`, use the following command:

```
>> prodserver.addon.install('twoFractals','localhost',9910)
```

- 6 Finally, invoke the `twoFractals` proxy function:

```
>> twoFractals(300,600,5)
```

Two windows appear, one containing the Mandelbrot set and one displaying the Koch snowflake.

## See Also

### More About

- “Execute Deployed MATLAB Functions” on page 9-5
- “Execute Deployed Functions Using HTTPS” on page 9-17
- “Configure Client-Server Communication” on page 9-11



# MATLAB Client Functions

---

## prodserver.addon.accessTokenPolicy

Set access token policy for user authorization

### Syntax

```
prodserver.addon.accessTokenPolicy(host,port,token,Name,Value)
prodserver.addon.accessTokenPolicy(host,port,token)
```

### Description

`prodserver.addon.accessTokenPolicy(host,port,token,Name,Value)` sets the Azure Active Directory (Azure AD) credentials for user authorization using one or more name-value arguments and sets a token generation policy to authorize a user that is using MATLAB Production Server add-ons to communicate with a server running at `host:port`.

This function requires MATLAB Client for MATLAB Production Server.

`prodserver.addon.accessTokenPolicy(host,port,token)` sets a token generation policy or sets the value of a bearer token to authorize a user that is using MATLAB Production Server add-ons to communicate with a server running at `host:port`.

### Examples

#### Use System-Generated Bearer Token

To use a system-generated bearer token, you must set the Azure AD app registration credentials using name-value arguments.

First, make sure that access control is enabled on the server. For more information, see “Application Access Control”.

Make sure that the MATLAB Production Server add-on of the deployed application is installed on the client machine. For more information about installing add-ons, see “Execute Deployed MATLAB Functions” on page 9-5.

Set the system to automatically generate the bearer token to use in requests to a server running at `localhost` and port 57142, and also specify Azure AD app registration credentials for user authorization.

```
prodserver.addon.accessTokenPolicy('localhost',57142,'automatic',...
'ClientID','0d912326-e439-41d0-822c-b15asdf6137c3',...
'ServerID','dwe4581bf-7867-4b90-a05a-16be6a82flkh',...
'IssuerURI','https://login.microsoftonline.com/yourcompany.com')
```

Typically, you set the Azure AD app registration credentials once per server instance.

## Specify Bearer Token

Specify a bearer token to use when communicating with a server.

First, enable access control on the server. For more information, see “Application Access Control”.

Make sure that the MATLAB Production Server add-on of the deployed application is installed on the client machine. For more information about installing add-ons, see “Execute Deployed MATLAB Functions” on page 9-5.

Specify the bearer token 'bearer\_token\_value' to use in requests to a server running at IP address 10.2.2.5 and port 57142.

```
prodserver.addon.accessTokenPolicy('10.2.2.5',57142,'bearer_token_value')
```

## Do Not Generate Token

Specify that no bearer token is required when access control is not enabled on a server.

Make sure that the MATLAB Production Server add-on of the deployed application is installed on the client machine. For more information about installing add-ons, see “Execute Deployed MATLAB Functions” on page 9-5.

Set the system to not generate a bearer token to use in requests to a server running at IP address 10.2.2.5 and port 57142.

```
prodserver.addon.accessTokenPolicy('10.2.2.5',57142,'none')
```

## Input Arguments

### host — Host name of server

character vector | string scalar

Host name of the server hosting a deployable archive from which the add-on is installed, specified as a character vector or string scalar.

Example: '144.213.5.7'

Data Types: char | string

### port — Port number of server

positive scalar

Port number of the server hosting a deployable archive from which the add-on is installed, specified as a positive scalar.

Example: 9920

Data Types: uint8 | uint16

### token — Access token policy

'none' (default) | 'automatic' | character vector | string scalar

Access token policy, specified as a character vector or string scalar. Set a token generation policy or specify a bearer token to authorize a user when communicating with a server. Possible options follow:

- 'automatic' — Generate bearer tokens using user credentials of the user logged in to the client machine. Azure AD app registration credentials must be set to use this policy.
- 'none' — Do not generate an access token. This value is the default.
- Character vector or string scalar — Specify a value to use as the bearer token.

If access control is enabled on the server, you must set the policy to 'automatic' or specify a bearer token.

Example: 'automatic'

Example: 'none'

Example: 'AAAAAAAAAABBBBBAAAAAAMLheAAAAAA0%2BuSep1%2BULvsea4JtiGRiSDSJSI%3DEUifiRmndf5E2XzMDjRfl76ZC9Ub0wnz4XsNiRVBChTYbJcE3F'

Data Types: char | string

### Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Example: 'ClientID', 'd17lk1bf-7977-4c90-a95a-16by7982fbaf', 'ServerID', 'd7pj91bf-7977-4b90-a05a-17vy5s82fbaf', 'IssuerURI', 'https://login.microsoftonline.com/your\_organization\_tenantID'

#### **ClientID** — Application ID of Azure AD client app

character vector | string scalar

Application ID of the client app registered in Azure AD used for user authorization, specified as a character vector or string scalar.

Example: 'ClientID', 'd17lk1bf-7977-4c90-a95a-16by7982fbaf'

Data Types: char | string

#### **ServerID** — Application ID of the Azure AD server app

character vector | string scalar

Application ID of the server app registered in Azure AD used for user authorization, specified as a character vector or string scalar.

Example: 'ServerID', 'd7pj91bf-7977-4b90-a05a-17vy5s82fbaf'

Data Types: char | string

#### **IssuerURI** — URI to generate token

character vector | string scalar

URI to generate a bearer token, specified as a character vector or string scalar. For Azure AD, the **IssuerURI** is `https://login.microsoftonline.com/` followed by the Azure AD tenant ID.

Example: 'IssuerURI', 'https://login.microsoftonline.com/your\_organization\_tenantID'

Data Types: char | string



## **See Also**

`prodserver.addon.set`

## **Topics**

“Application Access Control” on page 9-14

“Execute MATLAB Functions Using HTTPS”

**Introduced in R2020b**

## prodserver.addon.availableAddOns

MATLAB Production Server add-ons available on active server instance

### Syntax

```
addons = prodserver.addon.availableAddOns(host,port)
```

### Description

`addons = prodserver.addon.availableAddOns(host,port)` returns the add-ons available on an active MATLAB Production Server server instance.

This function requires MATLAB Client for MATLAB Production Server.

### Examples

#### Add-Ons Available on Server

Find the names of add-ons that are available on an active server instance.

First, host a deployable archive `fractal` on a MATLAB Production Server instance. You must include a MATLAB function signature file when you create the archive. You must enable the discovery service on the server instance that hosts the archive. For information on how to create and deploy the archive, see “Create Deployable Archive for MATLAB Production Server” on page 2-2 and “Share Deployable Archive”.

Start the server instance running at `localhost` and port 57142.

Find which add-ons are available on the server.

```
addons = prodserver.addon.availableAddOns('localhost',57142)
```

```
addons =
```

```
1×8 table
```

Name	Release	Version	Installed	Identifier
"fractal"	"R2020b"	"1.0.0"	true	"76643195-2ba3-4574-8fd0-084cc51251aa"

The output indicates that the `fractal` add-on is available and is also installed on the client machine.

### Input Arguments

#### host — Host name of server

character vector | string scalar

Host name of the server hosting a deployable archive from which an add-on is installed, specified as a character vector or string scalar.

Example: '144.213.5.7'

Data Types: char | string

### **port — Port number of server**

positive scalar

Port number of the server hosting a deployable archive from which an add-on is installed, specified as a positive scalar.

Example: 9920

Data Types: uint8 | uint16

## **Output Arguments**

### **addons — List of available MATLAB Production Server add-ons**

table

List of available MATLAB Production Server add-ons on a server instance, specified as a table. If multiple add-ons are available, each add-on is listed in a separate row. Each row has the following columns:

- **Name** — Name of the available add-on.
- **Release** — Version of MATLAB used to create the add-on.
- **Version** — Author-specified version of the add-on.
- **Installed** — Boolean indicating whether the add-on is installed on the client machine. If `true`, the add-on is installed. If `false`, the add-on is not installed.
- **Identifier** — String uniquely identifying the add-on. For more information, see `prodserver.addon.install`.
- **Host** — Host name of the server that makes the add-on available.
- **Port** — Port number of the server that makes the add-on available.

### **See Also**

`prodserver.addon.install` | `prodserver.addon.installFolder` |  
`prodserver.addon.isInstalled` | `prodserver.addon.uninstall`

**Introduced in R2019b**

## prodserver.addon.Explorer

Launch MATLAB Production Server Add-On Explorer app

### Syntax

```
prodserver.addon.Explorer
```

### Description

This function requires MATLAB Client for MATLAB Production Server.

`prodserver.addon.Explorer` opens the **MATLAB Production Server Add-On Explorer** app. The **MATLAB Production Server Add-On Explorer** app lets you add server instances that your MATLAB session can communicate with, and lets you browse, install, and uninstall MATLAB Production Server add-ons that communicate with the servers.

### Examples

#### Open MATLAB Production Server Add-On Explorer app

Open the **MATLAB Production Server Add-On Explorer** app.

```
prodserver.addon.Explorer
```

### See Also

```
prodserver.addon.install | prodserver.addon.uninstall
```

### Topics

“Execute Deployed MATLAB Functions” on page 9-5

**Introduced in R2019b**

# prodserver.addon.get

Get value of MATLAB Production Server add-on property

## Syntax

```
value = prodserver.addon.get(name)  
props = prodserver.addon.get
```

## Description

This function requires MATLAB Client for MATLAB Production Server.

`value = prodserver.addon.get(name)` returns the value of the MATLAB Production Server add-on property specified by name.

`props = prodserver.addon.get` returns the values of all MATLAB Production Server add-on properties.

## Examples

### Determine if HTTPS is Used for Client-Server Communication

Get the value of the `TransportLayerSecurity` property.

```
tls = prodserver.addon.get('TransportLayerSecurity')  
  
tls =  
  
    logical  
  
    0
```

The output indicates that the client does not use HTTPS for client-server communication.

### Get Values of All Properties

Get values of all MATLAB Production Server add-on properties.

```
props = prodserver.addon.get  
  
props =  
  
    struct with fields:  
  
        TransportLayerSecurity: 1  
        CertificateFile: [1x0 string]
```

The `TransportLayerSecurity` field in the output indicates that the client uses HTTPS for client-server communication.

## Input Arguments

### **name** — Name of MATLAB Production Server add-on property

character vector | scalar

Name of the MATLAB Production Server add-on property, specified as a character vector or scalar. You can specify only valid property names. Valid property names are:

- `TransportLayerSecurity`
- `CertificateFile`

Example: `'TransportLayerSecurity'`

Data Types: `char` | `string`

## Output Arguments

### **value** — Value of add-on property

character vector | string

Value of the MATLAB Production Server add-on property, returned as a character vector or string.

### **props** — Values of all add-on properties

structure

Values of all MATLAB Production Server add-on properties, returned as a structure. The structure contains the following fields:

- `TransportLayerSecurity` — Boolean that indicates whether the client uses HTTPS.
- `CertificateFile` — Path to the SSL certificate of the server. For more information, see “Save SSL Certificate of Server” on page 9-17.

## See Also

`prodserver.addon.set`

### External Websites

“Execute MATLAB Functions Using HTTPS”

### Introduced in R2020a

# prodserver.addon.install

Install MATLAB Production Server add-on from server

## Syntax

```
prodserver.addon.install(name,host,port)
prodserver.addon.install(name,host,port,'TransportLayerSecurity',tf)
prodserver.addon.install(name,endpoint)
info = prodserver.addon.install( ___ )
```

## Description

`prodserver.addon.install(name,host,port)` installs the MATLAB Production Server add-on name from a MATLAB Production Server instance running at `host` and `port`.

This function requires MATLAB Client for MATLAB Production Server.

`prodserver.addon.install(name,host,port,'TransportLayerSecurity',tf)` additionally lets you specify the URI scheme (HTTP or HTTPS) of the sever.

`prodserver.addon.install(name,endpoint)` lets you specify the network address for the active server instance from which you can install add-ons.

`info = prodserver.addon.install( ___ )` additionally returns information about the installed add-on using any of the input arguments in previous syntaxes.

## Examples

### Install Add-On Using Host Name and Port of Server

Install a MATLAB Production Server add-on using the default `http` scheme.

First, host a deployable archive `fractal` on a MATLAB Production Server instance. You must include a MATLAB function signature file when you create the archive. You must enable the discovery service on the server instance that hosts the archive. For information on how to create and deploy the archive, see “Create Deployable Archive for MATLAB Production Server” on page 2-2 and “Share Deployable Archive”.

Install the `fractal` add-on from a server running at IP address 10.2.2.5 and port 57142.

```
prodserver.addon.install('fractal','10.2.2.5',57142)
```

```
ans =
```

```
1×5 table
```

Name	Version	Enabled	Identifier
------	---------	---------	------------

```
"fractal (R2020b)"    "1.0.0"    true    "599ce38c-eea1-4011-85b9-8d301b4d5375"    "http:"
```

### Install Add-On Using Network Address of Server

Install a MATLAB Production Server add-on using the network address of the server.

First, host a deployable archive `fractal` on a MATLAB Production Server instance. You must include a MATLAB function signature file when you create the archive. You must enable the discovery service on the server instance that hosts the archive. For information on how to create and deploy the archive, see “Create Deployable Archive for MATLAB Production Server” on page 2-2 and “Share Deployable Archive”.

Install the `fractal` add-on from a server running at IP address 10.2.2.5 and port 57142.

```
prodserver.addon.install('fractal','http://10.2.2.5:57142')
```

```
ans =
```

```
1x5 table
```

Name	Version	Enabled	Identifier	
"fractal (R2020b)"	"1.0.0"	true	"599ce38c-eea1-4011-85b9-8d301b4d5375"	"http:"

### Install Add-On Using HTTPS

Use HTTPS for client-server communication when installing an add-on.

First, enable HTTPS on a MATLAB Production Server instance. For more information, see “Enable HTTPS”.

Host a deployable archive `fractal` on the server instance. You must include a MATLAB function signature file when you create the archive. You must enable the discovery service on the server instance that hosts the archive. For information on how to create and deploy the archive, see “Create Deployable Archive for MATLAB Production Server” on page 2-2 and “Share Deployable Archive”.

Install the `fractal` add-on from a server running at `https://10.2.2.5:57142`.

```
prodserver.addon.install('fractal','10.2.2.5',57143,'TransportLayerSecurity',true)
```

```
ans =
```

```
1x5 table
```

Name	Version	Enabled	Identifier	
"fractal (R2020b)"	"1.0.0"	true	"599ce38c-eea1-4011-85b9-8d301b4d5375"	"https"



## Get Add-On Information After Installing Add-On

Install an add-on and obtain information about the installed add-on.

First, host a deployable archive `fractal` on a MATLAB Production Server instance. You must include a MATLAB function signature file when you create the archive. You must enable the discovery service on the server instance that hosts the archive. For information on how to create and deploy the archive, see “Create Deployable Archive for MATLAB Production Server” on page 2-2 and “Share Deployable Archive”.

Save information to a table `info` when installing the `fractal` add-on from a server running at IP address 10.2.2.5 and port 57142.

```
info = prodserver.addon.install('fractal','10.2.2.5',57142)
```

```
info =
```

```
1x5 table
```

Name	Version	Enabled	Identifier	
"fractal (R2020b)"	"1.0.0"	true	"599ce38c-eea1-4011-85b9-8d301b4d5375"	"http:"

The table `info` contains information about the installed add-on `fractal`.

## Input Arguments

### **name** — Name of MATLAB Production Server add-on

character vector | string scalar

Name of the MATLAB Production Server add-on to install from a server, specified as a character vector or string scalar.

Example: 'fractal'

Data Types: char | string

### **endpoint** — Network address of server

character vector | string scalar

Network address of the server hosting a deployable archive from which the add-on is installed, specified as a character vector or string scalar. The network address has the format `scheme://host_name_of_server:port_number`.

Example: 'https://144.213.5.7:9920'

Data Types: char | string

### **host** — Host name of server

character vector | string scalar

Host name of the server hosting a deployable archive from which the add-on is installed, specified as a character vector or string scalar.

Example: '144.213.5.7'

Data Types: char | string

**port — Port number of server**

positive scalar

Port number of the server hosting a deployable archive from which the add-on is installed, specified as a positive scalar.

Example: 9920

Data Types: `uint8` | `uint16`**tf — Flag to set URI scheme**

false (default) | true

Flag that sets the URI scheme that the add-on uses when communicating with a server, specified as a logical scalar. If you do not set `tf` or if you set `tf` to `false`, the function uses `http`. If you set `tf` to `true`, the function uses `https`.

Example: `true`Data Types: `logical`

## Output Arguments

**info — Information about installed add-on**

table

Information about the installed add-on, returned as a table. The table contains the following columns:

- `Name` — Name of the installed add-on
- `Version` — Version of the installed add-on
- `Enabled` — Boolean indicating whether the add-on is available
- `Identifier` — String uniquely identifying the add-on
- `Endpoint` — Network address of the server hosting the add-on in the format `scheme://server_host_name:port_number`

## See Also

`prodserver.addon.availableAddOns` | `prodserver.addon.installFolder` |  
`prodserver.addon.isInstalled` | `prodserver.addon.uninstall`

### Topics

“Execute Deployed MATLAB Functions” on page 9-5

“Connect MATLAB Session to MATLAB Production Server” on page 9-2

**Introduced in R2019b**

# prodserver.addon.installFolder

Path to installation folder of MATLAB Production Server add-on

## Syntax

```
path = prodserver.addon.installFolder(name,host,port)
path = prodserver.addon.installFolder(name,host,
port,'TransportLayerSecurity',tf)
path = prodserver.addon.installFolder(name,endpoint)
```

## Description

`path = prodserver.addon.installFolder(name,host,port)` returns the full path to the folder on a local machine where a MATLAB Production Server add-on is installed. If the add-on is not present on the local machine, the function returns an empty string. The server does not have to be active for the function to return the path to the installation folder.

This function requires MATLAB Client for MATLAB Production Server.

`path = prodserver.addon.installFolder(name,host, port,'TransportLayerSecurity',tf)` additionally specifies a URI scheme (HTTP or HTTPS) when specifying the server address.

`path = prodserver.addon.installFolder(name,endpoint)` specifies a network address endpoint for the server instance.

## Examples

### Get Install Location of Add-On

Get the full path to an installed add-on from a server that uses the default HTTP scheme.

Install the `fractal` add-on from a server running at IP address 10.2.2.5 and port 57142. Get the path to the `fractal` add-on.

```
path = prodserver.addon.installFolder('fractal','10.2.2.5',57142)
```

```
path =
```

```
"C:\Users\username\AppData\Roaming\MathWorks\MATLAB Add-Ons\Toolboxes\fractal_ Hosted by MATI
```

### Get Install Location of Add-On Installed from Server Using HTTPS

Get the full path to an installed add-on from a server that uses the HTTPS scheme.

Install the `fractal` add-on from a server using HTTPS, and running at IP address 10.2.2.5 and port 57144. Get the path to the `fractal` add-on.

```
path = prodserver.addon.installFolder('fractal','10.2.2.5',57144,'TransportLayerSecurity',true)
path =
    'C:\Users\username\AppData\Roaming\MathWorks\MATLAB Add-Ons\Toolboxes\fractal_ Hosted by MATI
```

### Get Install Location of Add-On Using Network Address of Server

Get the full path to an installed add-on by specifying the network address of the server from which it was installed.

Install the `fractal` add-on from a server running at IP address 10.2.2.5 and port 57142.

Get the path to the `fractal` add-on.

```
path = prodserver.addon.installFolder('fractal','http://10.2.2.5:57142')
path =
    'C:\Users\username\AppData\Roaming\MathWorks\MATLAB Add-Ons\Toolboxes\fractal_ Hosted by MATI
```

## Input Arguments

### **name** — Name of MATLAB Production Server add-on

character vector | string scalar

Name of the MATLAB Production Server add-on, specified as a character vector or string scalar.

Example: `'fractal'`

Data Types: `char` | `string`

### **host** — Host name of server

character vector | string scalar

Host name of the server hosting a deployable archive from which the add-on is installed, specified as a character vector or string scalar.

Example: `'144.213.5.7'`

Data Types: `char` | `string`

### **port** — Port number of server

positive scalar

Port number of the server hosting a deployable archive from which the add-on is installed, specified as a positive scalar.

Example: `9920`

Data Types: `uint8` | `uint16`

### **endpoint** — Network address of server

character vector | string scalar

Network address of the server hosting a deployable archive from which the add-on is installed, specified as a character vector or string scalar. The network address has the format `scheme://host_name_of_server:port_number`.

Example: `'https://144.213.5.7:9920'`

Data Types: `char` | `string`

### **tf – Flag to set URI scheme**

`false` (default) | `true`

Flag to set the URI scheme that the add-on uses when communicating with a server, specified as a logical scalar. If you do not set `tf` or if you set `tf` to `false`, the function uses HTTP. If you set `tf` to `true`, the function uses HTTPS.

Example: `true`

Data Types: `logical`

### **See Also**

`prodserver.addon.availableAddOns` | `prodserver.addon.install` | `prodserver.addon.isInstalled`

**Introduced in R2020b**

## prodserver.addon.isAddOnFcn

Determine if function is installed as part of MATLAB Production Server add-on

### Syntax

```
tf = prodserver.addon.isAddOnFcn(fcn)
[tf,name] = prodserver.addon.isAddOnFcn(fcn)
```

### Description

`tf = prodserver.addon.isAddOnFcn(fcn)` returns logical 1 (`true`) if the function `fcn` is present on the MATLAB path and originates in a folder that belongs to a MATLAB Production Server add-on. Otherwise, the function returns logical 0 (`false`). The function returns `false` if the add-on containing `fcn` is disabled or not installed, because disabling an add-on removes its functions from the MATLAB path.

This function requires MATLAB Client for MATLAB Production Server.

`[tf,name] = prodserver.addon.isAddOnFcn(fcn)` additionally returns the name of the add-on that contains the function `fcn`. If the add-on containing `fcn` is disabled or not installed, the function returns an empty string.

### Examples

#### Find If Function Is Installed on Client Machine

Check if the function `mandelbrot` is installed on the client machine as part of an add-on.

```
tf = prodserver.addon.isAddOnFcn('mandelbrot')
```

```
tf =
```

```
    logical
```

```
    0
```

The output indicates that the function `mandelbrot` is not installed as a part of any add-on on the client machine.

#### Find Name of Add-On That Contains Function

Find which add-on contains a particular function.

Consider a deployable archive `fractal` that you host on the server instance. For more information on how to create and deploy an archive, see “Create Deployable Archive for MATLAB Production Server” on page 2-2 and “Share Deployable Archive”.

After you install the `fractal` add-on from the server instance, check if the function `mandelbrot` is present in the add-on.

```
[tf,name] = prodserver.addon.isAddonFcn('mandelbrot')  
  
tf =  
  
    logical  
  
     1  
  
name =  
  
    "fractal"
```

The output indicates that the function `mandelbrot` is installed on the client machine as part of the `fractal` add-on.

## Input Arguments

### **fcn** — Name of function

character vector | string scalar

Name of the function, specified as a character vector or string scalar.

Example: `'mandelbrot'`

Data Types: `char` | `string`

## Output Arguments

### **tf** — Value that indicates if function is installed as part of add-on

logical scalar

Value that indicates if the function is installed as part of an add-on, specified as a character vector or string scalar.

### **name** — Name of add-on that contains function

character vector | string scalar

Name of the add-on that contains the function, specified as a character vector or string scalar. If the add-on containing the function is disabled or not installed, the function returns an empty string.

## See Also

`prodserver.addon.availableAddOns` | `prodserver.addon.install` |  
`prodserver.addon.isInstalled` | `prodserver.addon.uninstall`

**Introduced in R2020b**

## prodserver.addon.isInstalled

Determine if MATLAB Production Server add-on is installed from server instance

### Syntax

```
tf = prodserver.addon.isInstalled(name,host,port)
tf = prodserver.addon.isInstalled(name,host,port,'TransportLayerSecurity',
setsecurity)
tf = prodserver.addon.isInstalled(name,endpoint)
[tf,tls] = prodserver.addon.isInstalled( ___ )
```

### Description

`tf = prodserver.addon.isInstalled(name,host,port)` returns logical 1 (true) if a MATLAB Production Server add-on name is installed from a server instance whose address is specified by `host` and `port`, and returns logical 0 (false) otherwise.

The server instance does not need to be active when you run this function.

This function requires MATLAB Client for MATLAB Production Server.

`tf = prodserver.addon.isInstalled(name,host,port,'TransportLayerSecurity',setsecurity)` additionally specifies a URI scheme (HTTP or HTTPS) for the server address.

`tf = prodserver.addon.isInstalled(name,endpoint)` specifies a network address for the server instance.

`[tf,tls] = prodserver.addon.isInstalled( ___ )` additionally returns the value of the `TransportLayerSecurity` property using any of the input arguments in the previous syntaxes.

### Examples

#### Determine If Add-On Is Installed by Specifying Host and Port of Server

After you install the `fractal` add-on from a server running at IP address 10.2.2.5 and port 57140, check if the add-on is installed from the server.

```
tf = prodserver.addon.isInstalled('fractal','10.2.2.5','57140')
```

```
tf =
    logical
     1
```

The output indicates that the `fractal` add-on is installed from the server specified by IP address 10.2.2.5 and port number 57140.



### Determine If Add-On Is Installed From Server Using HTTPS

After you install the `fractal` add-on from a server using HTTPS and running at IP address 10.2.2.5 and port 57140, check if the add-on is installed from the server.

```
tf = prodserver.addon.isInstalled('fractal','10.2.2.5','57142','TransportLayerSecurity',true)
```

```
tf =  
  
    logical  
  
     1
```

The output indicates that the `fractal` add-on is installed from the server specified by IP address 10.2.2.5 and port number 57142.

### Determine If Add-On Is Installed and if Server Uses HTTPS

After you install the `fractal` add-on from a server running at IP address 10.2.2.5 and port 57140, check if the add-on is installed from the server. Additionally, check if the server uses HTTPS.

```
[tf,tls] = prodserver.addon.isInstalled('fractal','http://10.2.2.5:57140')
```

```
tf =  
  
    logical  
  
     1  
  
tls =  
  
    logical  
  
     0
```

The output indicates that the `fractal` add-on is installed from the specified server and the scheme is HTTP.

## Input Arguments

### **name** — Name of MATLAB Production Server add-on

character vector | string scalar

Name of the MATLAB Production Server add-on, specified as a character vector or string scalar.

Example: `'fractal'`

Data Types: `char` | `string`

### **host** — Host name of server

character vector | string scalar

Host name of the server hosting a deployable archive from which you can install add-ons, specified as a character vector or string scalar.

Example: '144.213.5.7'

Data Types: char | string

#### **port** — Port number of server

positive scalar

Port number of the server hosting a deployable archive from which you can install add-ons, specified as a positive scalar.

Example: 9920

Data Types: uint8 | uint16

#### **endpoint** — Network address of server

character vector | string scalar

Network address of the server hosting a deployable archive from which you can install add-ons, specified as a character vector or string scalar. The network address has the format `scheme://host_name_of_server:port_number`.

Example: 'https://144.213.5.7:9920'

Data Types: char | string

#### **setsecurity** — Flag that sets URI scheme

false (default) | true

Flag that sets the URI scheme that the add-on uses when communicating with a server, specified as a logical scalar. If you do not set `setsecurity` or if you set `setsecurity` to false, the scheme is HTTP. If you set `setsecurity` to true, the scheme is HTTPS.

Example: 'TransportLayerSecurity', true

Data Types: logical

## Output Arguments

#### **tf** — Value that indicates whether function is installed

logical scalar

Value that indicates whether the function is installed, returned as a logical scalar.

#### **tls** — Value of TransportLayerSecurity property

logical scalar

Value of the `TransportLayerSecurity` property, returned as a logical scalar. If the value of `TransportLayerSecurity` is logical 1 (true), the client-server communication uses HTTPS; otherwise, it uses HTTP.

## See Also

`prodserver.addon.availableAddOns` | `prodserver.addon.install` |  
`prodserver.addon.installFolder` | `prodserver.addon.isAddOnFcn` |  
`prodserver.addon.set`

**Introduced in R2019b**

## prodserver.addon.set

Set value of MATLAB Production Server add-on property

### Syntax

```
prodserver.addon.set('TransportLayerSecurity',tf)
prodserver.addon.set('CertificateFile',path)
prodserver.addon.set('TransportLayerSecurity',tf,'CertificateFile',path)
```

### Description

`prodserver.addon.set('TransportLayerSecurity',tf)` sets the client-server communication to use HTTPS or HTTP by setting the value of the `TransportLayerSecurity` property. If the value of `tf` is true, the client-server communication uses HTTPS; otherwise, it uses HTTP.

This function requires MATLAB Client for MATLAB Production Server.

`prodserver.addon.set('CertificateFile',path)` sets the path to the SSL certificate of the server that is saved on the client machine by setting the `CertificateFile` property. You might need to set the path if you save the self-signed SSL certificate of the server locally or for other testing purposes.

`prodserver.addon.set('TransportLayerSecurity',tf,'CertificateFile',path)` lets you set the `TransportLayerSecurity` and `CertificateFile` properties at the same time.

### Examples

#### Use HTTPS for Client-Server Communication

Use HTTPS when communicating with server instances by setting the `TransportLayerSecurity` property to true.

```
prodserver.addon.set('TransportLayerSecurity',true)
```

This setting persists across MATLAB sessions.

#### Set Path to Self-Signed Server Certificate

Before your client can communicate with a server that uses a self-signed SSL certificate, you must save the server certificate locally. For more information, see “Save SSL Certificate of Server” on page 9-17.

Then, set the path to the server certificate that you saved.

```
prodserver.addon.set('CertificateFile','C:\server_cert.pem')
```

This setting persists across MATLAB sessions.

## **See Also**

`prodserver.addon.get`

## **Topics**

“Execute MATLAB Functions Using HTTPS”

**Introduced in R2020a**

## prodserver.addon.uninstall

Uninstall MATLAB Production Server add-on

### Syntax

```
prodserver.addon.uninstall(name,host,port)
prodserver.addon.uninstall(name,host,port,'TransportLayerSecurity',tf)
prodserver.addon.uninstall(name,endpoint)
prodserver.addon.uninstall(identifier)
```

### Description

`prodserver.addon.uninstall(name,host,port)` uninstalls a MATLAB Production Server add-on that is installed from a server whose address is specified by `host` and `port`.

Uninstalling a MATLAB Production Server add-on removes the add-on, including all functions, examples, and documentation available in the add-on. Uninstalling a MATLAB Production Server add-on does not modify or remove any code that calls the functions available in the add-on. The server instance is not required to be running for the add-on to be uninstalled.

This function requires MATLAB Client for MATLAB Production Server.

`prodserver.addon.uninstall(name,host,port,'TransportLayerSecurity',tf)` additionally allows you to specify the URI scheme (HTTP or HTTPS) that the server uses.

`prodserver.addon.uninstall(name,endpoint)` lets you specify a network address to identify the server.

`prodserver.addon.uninstall(identifier)` uninstalls the add-on specified by `identifier`.

### Examples

#### Uninstall Add-On Using Add-On Name, and Server Host Name and Port

Uninstall a MATLAB Production Server add-on using the add-on name, and the host name and port number of the server from which the add-on was installed.

First, install the `fractal` add-on from a server running at IP address 10.2.2.5 and port 57142. For more information, see `prodserver.addon.install`.

Use the name of the add-on, and the IP address and port number of the server from which it was installed to uninstall the add-on.

```
prodserver.addon.uninstall('fractal','10.2.2.5',57142)
```

## Specify HTTPS Scheme When Uninstalling Add-On

Uninstall a MATLAB Production Server add-on that uses HTTPS to communicate with a server by using the host name and port number of the server from which it was installed.

First, install the `fractal` add-on from a server running at IP address 10.2.2.5 and port 57144, and using the default HTTP scheme. For more information, see `prodserver.addon.install`.

To uninstall the add-on, specify the name of the add-on, and the host name and port number of the server from which the add-on was installed. Also, specify the HTTPS scheme by setting the `TransportLayerSecurity` property to `true`.

```
prodserver.addon.uninstall('fractal','10.2.2.5',57144,'TransportLayerSecurity',true)
```

## Uninstall Add-On Using Add-On Name and Network Address of Server

Uninstall a MATLAB Production Server add-on using the add-on name and network address of the server from which it was installed.

First, install the `fractal` add-on from a server running at `http://10.2.2.5:57142`. For more information, see `prodserver.addon.install`.

Use the name of the add-on and the network address of the server from which it was installed to uninstall the add-on.

```
prodserver.addon.uninstall('fractal','http://10.2.2.5:57142')
```

## Uninstall Add-On Using Unique Identifier

Uninstall a MATLAB Production Server add-on using its unique identifier.

First, install a MATLAB Production Server add-on `fractal` using `prodserver.addon.install`.

`prodserver.addon.install` returns an identifier that uniquely identifies the add-on.

Use the unique identifier to uninstall the add-on.

```
prodserver.addon.uninstall('3c192cbd-95dc-4263-a722-6d594b9ae12c')
```

## Input Arguments

### **identifier** — String uniquely identifying the add-on

character vector | string scalar

String uniquely identifying the add-on, specified as a character vector or string scalar.

Example: `'3c192cbd-95dc-4263-a722-6d594b9ae12c'`

Data Types: `char` | `string`

### **name** — Name of MATLAB Production Server add-on

character vector | string scalar

Name of the MATLAB Production Server add-on to uninstall, specified as a character vector or string scalar.

Example: 'fractal'

Data Types: char | string

### **endpoint** — Network address of server

character vector | string scalar

Network address of the server hosting a deployable archive from which the add-on is installed, specified as a character vector or string scalar. The network address has the format `scheme://host_name_of_server:port_number`.

Example: 'https://localhost:9910'

Data Types: char | string

### **host** — Host name of server

character vector | string scalar

Host name of the server hosting a deployable archive from which the add-on is installed, specified as a character vector or string scalar.

Example: '144.213.5.7'

Data Types: char | string

### **port** — Port number of server

positive scalar

Port number of the server hosting a deployable archive from which the add-on is installed, specified as a positive scalar.

Example: 9920

Data Types: uint8 | uint16

### **tf** — Flag to determine URI scheme

false (default) | true

Flag that determines the URI scheme that the server uses, specified as a logical scalar.

- `true` — The add-on uses HTTPS.
- `false` — The add-on uses HTTP.

Example: 'TransportLayerSecurity',true

Data Types: logical

## **See Also**

`prodserver.addon.install`

**Introduced in R2019b**



# MATLAB Client Apps

---

