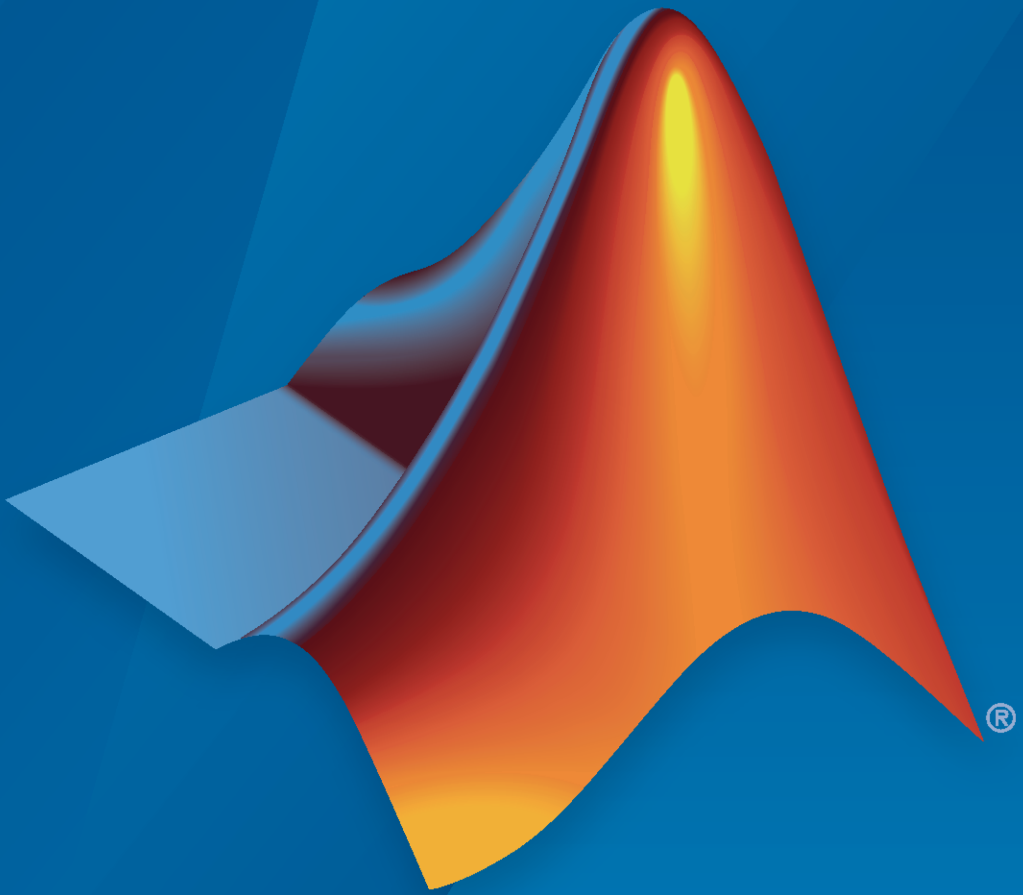
MATLAB® Production Server™

Code Deployment



MATLAB®

R2023a

MathWorks®

How to Contact MathWorks

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

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

MATLAB® Production Server™ Code Deployment

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)
September 2021	Online only	Revised for Version 4.6 (Release R2021b)
March 2022	Online only	Revised for Version 5.0 (Release R2022a)
September 2022	Online only	Revised for Version 5.1 (Release R2022b)
March 2023	Online only	Revised for Version 5.2 (Release R2023a)

Contents

1 Create a Deployable Archive from MATLAB Production Server Code

Create Deployable Archive for MATLAB Production Server **1-2**
Create MATLAB Function **1-2**
Create Deployable Archive with Production Server Compiler App **1-2**
Customize Application and Its Appearance **1-3**
Package Application **1-4**
Create Deployable Archive Using compiler.build.productionServerArchive **1-5**
Compatibility Considerations **1-5**

Package Deployable Archives with Production Server Compiler App **1-7**
Create Function In MATLAB **1-7**
Create Deployable Archive with Production Server Compiler App **1-7**
Customize the Application and Its Appearance **1-8**
Package the Application **1-8**

Package Deployable Archives from Command Line **1-10**
Execute Compiler Projects with deploytool **1-10**
Package a Deployable Archive with mcc **1-10**
Differences Between Compiler Apps and Command Line **1-10**

Modifying Deployed Functions **1-12**

Use Parallel Computing Resources in Deployable Archives **1-13**
Use Profile Available in Cluster Profile Manager **1-13**
Link to Exported Profile **1-13**
Reuse Existing Parallel Pool in Deployable Archive **1-14**
Limitations **1-14**

2 Customizing a Compiler Project

Customize an Application **2-2**
Customize the Installer **2-2**
Manage Required Files in Compiler Project **2-4**
Sample Driver File Creation **2-5**
Specify Files to Install with Application **2-6**
Additional Runtime Settings **2-6**

Manage Support Packages **2-7**
Using a Compiler App **2-7**

Using the Command Line **2-7**

3 Advanced Uses of the Command Line Compiler

Simplify Compilation Using Macros **3-2**
Macros **3-2**
Working With Macros **3-2**

Invoke MATLAB Build Options **3-4**
Specify Full Path Names to Build MATLAB Code **3-4**
Using Bundles to Build MATLAB Code **3-4**

MATLAB Runtime Component Cache and Deployable Archive Embedding **3-6**
Overriding Default Behavior **3-7**
For More Information **3-7**

mcc Command Arguments Listed Alphabetically **3-8**
Packaging Log and Output Folders **3-10**

4 Functions

5 Apps

6 Persistence

Data Caching Basics **6-2**
Typical Workflow for Data Caching **6-2**
Configure Server to Use Redis **6-2**
Example: Increment Counter Using Data Cache **6-3**

Manage Application State in Deployed Archives **6-5**
Step 1: Write MATLAB Code that uses Persistence Functions **6-5**
Step 2: Run Example in Testing Workflow **6-9**
Step 3: Run Example in Deployment Workflow **6-10**

Handle Custom Routes and Payloads in HTTP Requests **6-14**
Write MATLAB Function for Web Request Handler **6-14**
Configure Server for URL Routes **6-15**

End-to-End Setup for Web Request Handler **6-16**

7 Persistence Functions

8 MATLAB Client

Connect MATLAB Session to MATLAB Production Server **8-2**
When to Use MATLAB Client for MATLAB Production Server **8-2**
Install MATLAB Client for MATLAB Production Server **8-2**
Connect MATLAB Session to MATLAB Production Server **8-2**
System Requirements **8-3**
Synchronous Function Execution **8-3**
Supported Data Types **8-3**

Execute Deployed MATLAB Functions **8-5**
Install MATLAB Client for MATLAB Production Server **8-5**
Deploy MATLAB Function on Server **8-5**
Install MATLAB Production Server Add-On for the Deployable Archive . . . **8-6**
Manage Installed Add-On **8-8**
Invoke Deployed MATLAB Function **8-9**

Configure Client-Server Communication **8-11**
Configure Timeouts and Retries **8-11**
Update Server Configuration **8-12**

Application Access Control **8-14**
Prerequisites **8-14**
Configure Access Control **8-14**

Execute Deployed Functions Using HTTPS **8-17**
Save SSL Certificate of Server **8-17**
Install Add-On Using HTTPS **8-18**
Manage Default Protocol for Client-Server Communication **8-18**

Manage Add-Ons **8-20**
Install Add-Ons **8-20**
Remove Add-Ons **8-22**
Get Information about Add-Ons **8-22**
Manage Add-Ons **8-23**
Manage Access to Applications Deployed on Server **8-23**

Deploy Add-Ons **8-25**
Prerequisites **8-26**
Create Standalone Executables That Use Add-Ons **8-26**
Create Shared Libraries or Software Components That Use Add-Ons . . . **8-27**
Create Deployable Archives That Use Add-Ons **8-28**

9 MATLAB Client Functions

10 Streaming Functions

11 Streaming Topics

Streaming Data Framework for MATLAB Production Server Basics 11-2
Install Streaming Data Framework for MATLAB Production Server **11-2**
System Requirements **11-2**
Write Streaming Analytic MATLAB Function **11-2**
Process Kafka Events Using MATLAB **11-4**
Simulate Production Using Development Version of MATLAB Production Server **11-4**
Deploy Streaming Analytic to MATLAB Production Server **11-4**

Process Kafka Events Using MATLAB 11-5
Prerequisites **11-5**
Write Streaming Analytic MATLAB Function **11-5**
Create Sample Stream Events **11-6**
Validate Sample Data Creation **11-7**
Process Stream Events with Streaming Analytic Function **11-7**

Connect to Secure Kafka Cluster 11-9
Kafka Provider Properties **11-9**
Connect to Secure Kafka Cluster **11-9**

Test Streaming Analytic Function Using Local Test Server 11-12
Prerequisites **11-12**
Write Streaming Analytic MATLAB Function **11-12**
Create Sample Streaming Data **11-12**
Simulate Production Using Local Test Server **11-13**

Deploy Streaming Analytic Function to MATLAB Production Server . . 11-17
Kafka Connector Specifications **11-17**
Prerequisites for Running Example **11-17**
Write Streaming Analytic MATLAB Function **11-18**
Package Streaming Analytic Function **11-18**
Deploy Streaming Analytic Function to Server **11-19**
Start Kafka Connector **11-19**
Read Processed Data From Output Stream **11-19**
Stop Kafka Connector **11-20**

Obtain Kafka Event Stream Log Files 11-21
Configure kafkaStream Object Logging **11-21**
Configure librdkafka Library Logging **11-22**

Configure Logging in Deployed Applications **11-22**
Provide Log Files to MathWorks Technical Support **11-23**

Create a Deployable Archive from MATLAB Production Server Code

- "Create Deployable Archive for MATLAB Production Server" on page 1-2
- "Package Deployable Archives with Production Server Compiler App" on page 1-7
- "Package Deployable Archives from Command Line" on page 1-10
- "Modifying Deployed Functions" on page 1-12
- "Use Parallel Computing Resources in Deployable Archives" on page 1-13

Create Deployable Archive for MATLAB Production Server

Supported platform: Windows®, Linux®, Mac

Note To create a deployable archive, you need an installation of the MATLAB Compiler SDK™ product.

This example shows how to create a deployable archive using a MATLAB function. You can then deploy the generated archive on MATLAB Production Server.

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

Package the function into a deployable archive using the Production Server Compiler app. Alternatively, if you want to create a deployable archive from the MATLAB command window using a programmatic approach, see “Create Deployable Archive Using compiler.build.productionServerArchive” (MATLAB Compiler SDK).

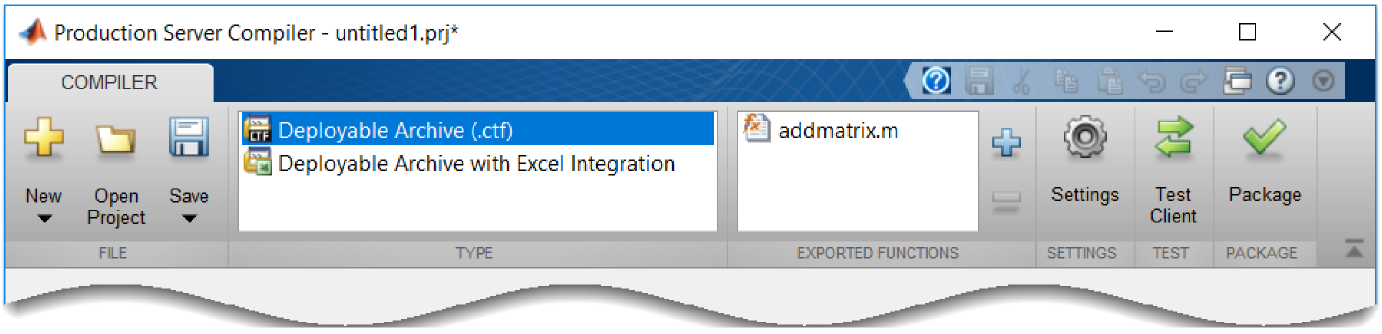
1. To open the **Production Server Compiler** app, type `productionServerCompiler` at the MATLAB prompt.

   Alternatively, 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)**.
2. In the **Production Server Compiler** 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**.

Doing so adds the function `addmatrix.m` to the list of main files.


Production Server Compiler - untitled1.prj*
COMPILER
New
Open Project
Save
FILE
Deployable Archive (.ctf)
Deployable Archive with Excel Integration
TYPE
addmatrix.m
EXPORTED FUNCTIONS
Settings
SETTINGS
Test Client
TEST
Package
PACKAGE

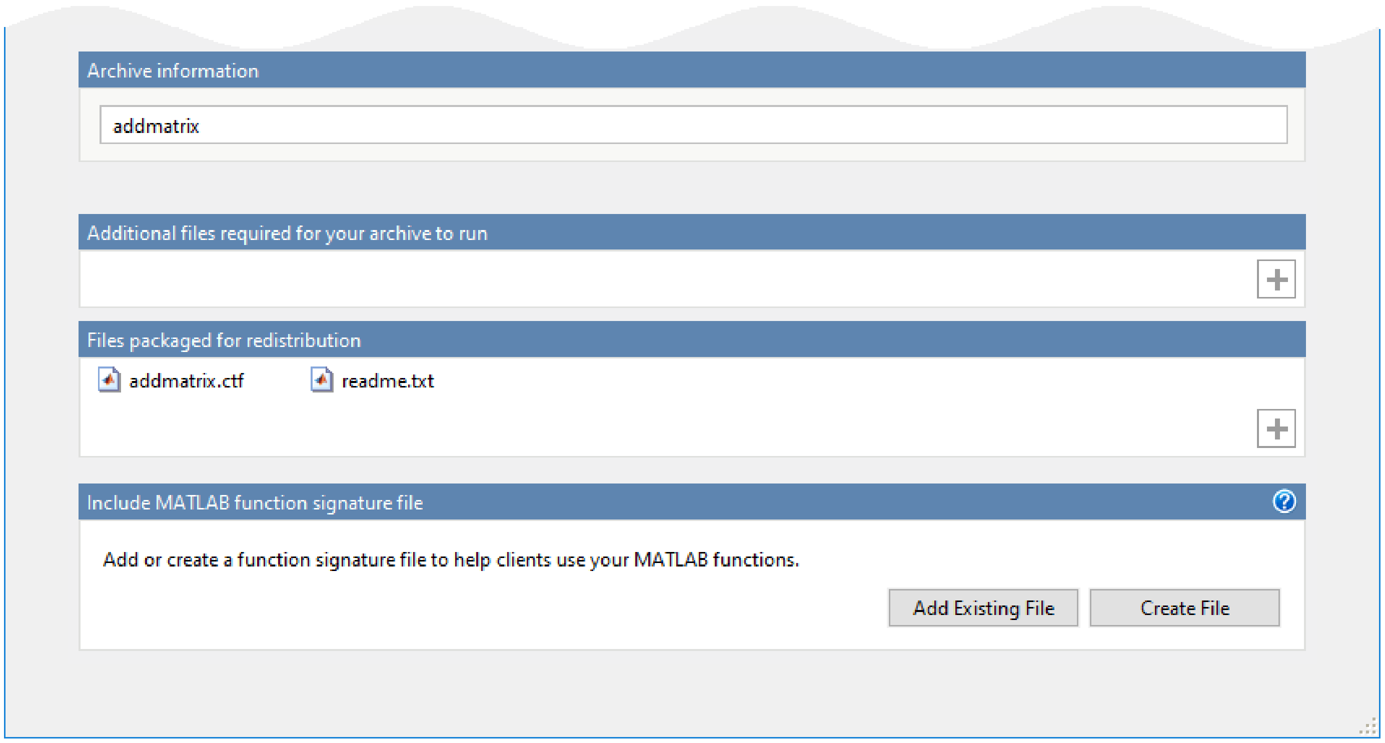
Customize Application and Its Appearance

Customize your deployable archive and add more information about the application.

- **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”.


Archive information

addmatrix

Additional files required for your archive to run

Files packaged for redistribution

addmatrix.ctf readme.txt

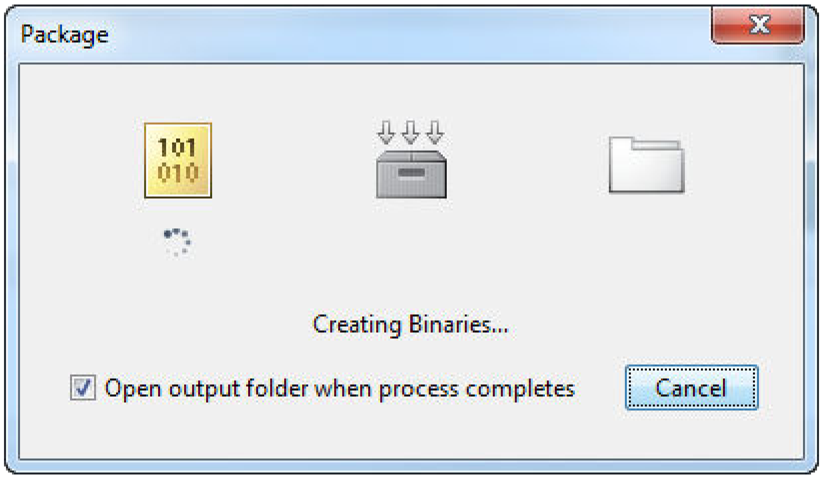
Include MATLAB function signature file

Add or create a function signature file to help clients use your MATLAB functions.

Package Application

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

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



Creating Binaries...

[✓] Open output folder when process completes

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.html` — Log file generated by MATLAB Compiler SDK

Create Deployable Archive Using compiler.build.productionServerArchive

As an alternative to the **Production Server Compiler** app, you can create a deployable archive using a programmatic approach.

- Build the deployable archive using the `compiler.build.productionServerArchive` function.

  Optionally, you can add a function signature file to help clients use your MATLAB functions. For more details, see “MATLAB Function Signatures in JSON”.

  ```
  buildResults = compiler.build.productionServerArchive('addmatrix.m',...
  'FunctionSignatures','addmatrixFunctionSignatures.json',...
  'Verbose','on');
  ```

  ```
  buildResults = 

    Results with properties:

                  BuildType: 'productionServerArchive'
                      Files: {'/home/mluser/Work/magicarchiveproductionServerArchive/addmatri
      IncludedSupportPackages: {}
                    Options: [1×1 compiler.build.ProductionServerArchiveOptions]
  ```

  You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.productionServerArchive`.

  The `compiler.build.Results` object `buildResults` contains information on the build type, generated files, included support packages, and build options.

  The function generates the following files within a folder named `addmatrixproductionServerArchive` in your current working directory:

  - `addmatrix.ctf` — Deployable archive file.
  - `includedSupportPackages.txt` — Text file that lists all support files included in the assembly.
  - `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` — Text file that contains packaging and deployment information.
  - `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
  - `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

Compatibility Considerations

In most cases, you can generate the deployable archive on one platform and deploy to a server running on any other supported platform. Unless you add operating system-specific dependencies or content, such as MEX files or Simulink® simulations to your applications, the generated archives are platform-independent.

See Also
`compiler.build.productionServerArchive` | `deploytool` | **Production Server Compiler** | `mcc`

More About
- “Test Client Data Integration Against MATLAB” (MATLAB Compiler SDK)
- Production Server Compiler
- “Deploy Archive to MATLAB Production Server”
- “MATLAB Function Signatures in JSON”
- “JSON Representation of MATLAB Data Types”

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;
```

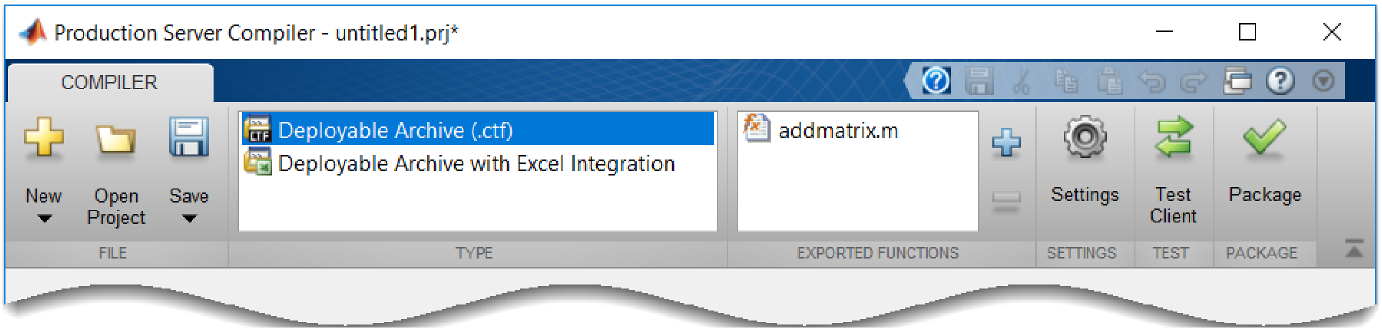
At the MATLAB command prompt, enter `addmatrix([1 4 7; 2 5 8; 3 6 9], [1 4 7; 2 5 8; 3 6 9])`.

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)**.



Deployable Archive (.ctf)
Deployable Archive with Excel Integration
addmatrix.m

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 toolstrip, click ➕.

   2 In the **Add Files** window, browse to the example folder, and select the function you want to package. Click **Open**.

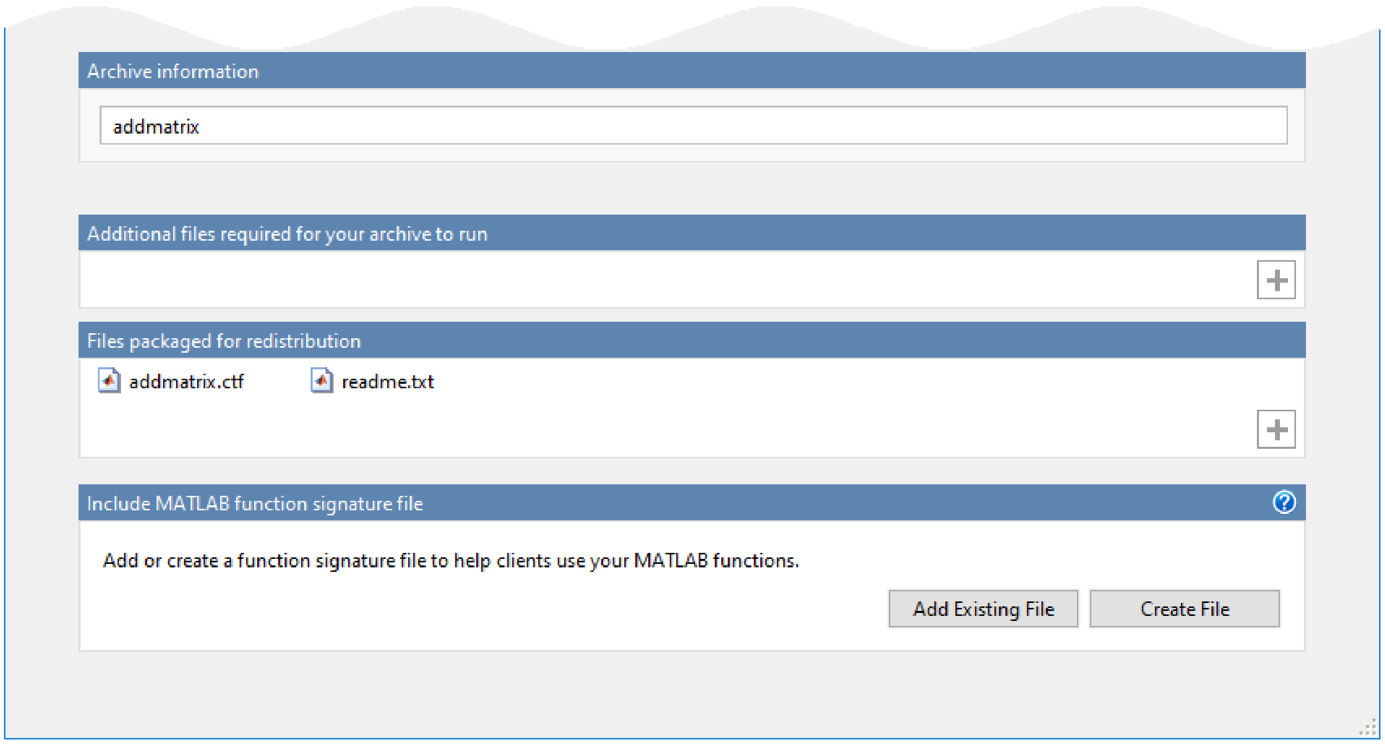
The function `addmatrix.m` is added to the list of main files.

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.


Archive information

addmatrix

Additional files required for your archive to run

Files packaged for redistribution

addmatrix.ctf readme.txt

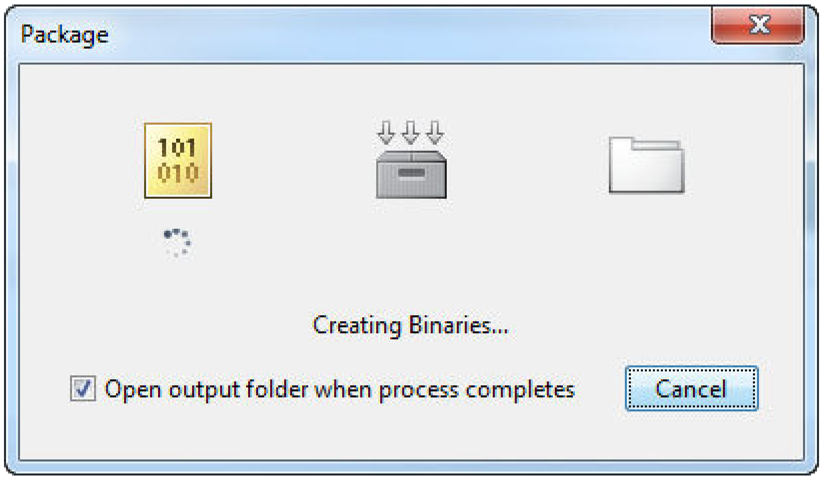
Include MATLAB function signature file

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

`productionServerCompiler` | `mcc` | `deploytool`

More About

- Production Server Compiler

Package Deployable Archives from Command Line

In this section...
"Execute Compiler Projects with deploytool" on page 1-10
"Package a Deployable Archive with mcc" on page 1-10
"Differences Between Compiler Apps and Command Line" on page 1-10

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
`-a` *`filePath`*	Add any files on the path to the generated binary.
`-d` *`outFolder`*	Specify the folder into which the results of packaging are written.
`class{`*`className`*`:`*`mfilename`*`...}`	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

`mcc` | `deploytool`

More About

- "Package Deployable Archives with Production Server Compiler App" on page 1-7

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

- "Deploy Archive to MATLAB Production Server"

Use Parallel Computing Resources in Deployable Archives

To take advantage of resources from Parallel Computing Toolbox, you can pass a cluster profile to a MATLAB application that you deploy to MATLAB Production Server.

Cluster profiles let you define parallel computing properties for your cluster, such as information about the cluster for your MATLAB code to use and the number of workers in a parallel pool. You apply these properties when you create a cluster, job, and task objects in your MATLAB application. For more information on specifying cluster profile preferences, see “Specify Your Parallel Preferences” (Parallel Computing Toolbox). To manage cluster profiles, see “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox).

You can also package MATLAB functions that use parallel language commands into a deployable archive and deploy the archive to MATLAB Production Server. For information on creating and sharing deployable archives, see “Create Deployable Archive for MATLAB Production Server” on page 1-2 and “Deploy Archive to MATLAB Production Server”.

Deployed MATLAB functions are able to find the parallel cluster profile through the Cluster Profile Manager or an exported profile.

Use Profile Available in Cluster Profile Manager

When you package a MATLAB function into a deployable archive, all profiles available in the Cluster Profile Manager are available in the archive by default. This option is useful when you do not expect the profile to change after deployment.

Link to Exported Profile

If you expect the cluster profile to change, you can export the cluster profile first, then load the profile either programmatically in your MATLAB code or use the `--user-data` MATLAB Production Server configuration property. For exporting the cluster profile, see “Import and Export Cluster Profiles” (Parallel Computing Toolbox).

Load Profile Using MATLAB Code

To load the exported profile in your MATLAB function, use `parallel.importProfile`. For example, the following sample code imports a profile and creates a cluster object using an exported profile.

```
clustername = parallel.importProfile('ServerIntegrationTest.settings');
cluster = parcluster(clustername);
```

Load Profile Using Server Configuration Property

To load the exported profile using the MATLAB Production Server configuration property, set the `--user-data` property to pass key-value parameters that represent the exported profile. Set the key to `ParallelProfile` and the value to the path to the exported cluster profile followed by the profile file name. For example, to load a profile called `ServerIntegrationTest.settings`, set the property as follows:

```
--user-data ParallelProfile /sandbox/server_integration/
ServerIntegrationTest.settings
```

If you use the command line to manage the dashboard, edit the `main_config` server configuration file to specify the `--user-data` property. If you use the dashboard to manage MATLAB Production Server, use the **Additional Data** field in the **Settings** tab to specify the `--user-data` property.

The cluster profile that you provide to the `--user-data` property is automatically set as the default. Therefore, your MATLAB code does not have to explicitly load it and you can use the default cluster as follows:

```
cluster = parcluster();
```

Reuse Existing Parallel Pool in Deployable Archive

The following example uses `gcp` to check if a parallel pool of workers exists. If a pool does not exist, it creates a pool of 4 workers using `parpool`.

```
pool = gcp('nocreate');
if isempty(pool)
    disp("Creating a myCluster")
    parpool('myCluster', 4);
else
    disp('myCluster pool already exists')
end
```

Limitations

Deployable archives that use parallel computing cannot share parallel pools with other deployable archives.

See Also

`parallel.importProfile` | `parallel.exportProfile` | `gcp` | `parpool`

Related Examples

- “Using MATLAB Runtime User Data Interface” (MATLAB Compiler SDK)
- “Create Deployable Archive for MATLAB Production Server” on page 1-2
- “Run MATLAB Parallel Server and MATLAB Production Server on Azure”

Customizing a Compiler Project

- "Customize an Application" on page 2-2
- "Manage Support Packages" on page 2-7

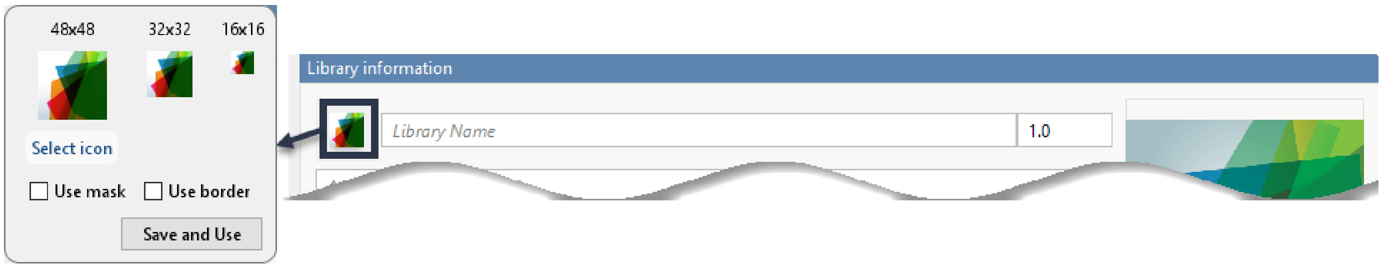
Customize an Application

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**.

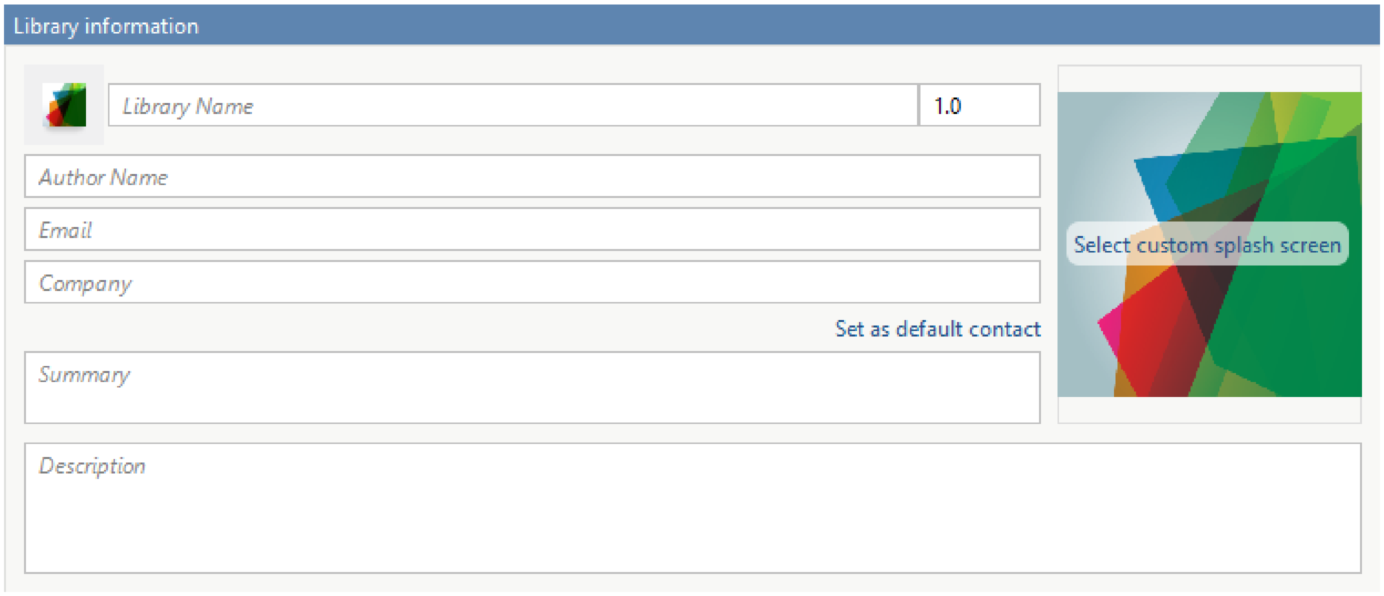
Add Library or Application Information

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.



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.

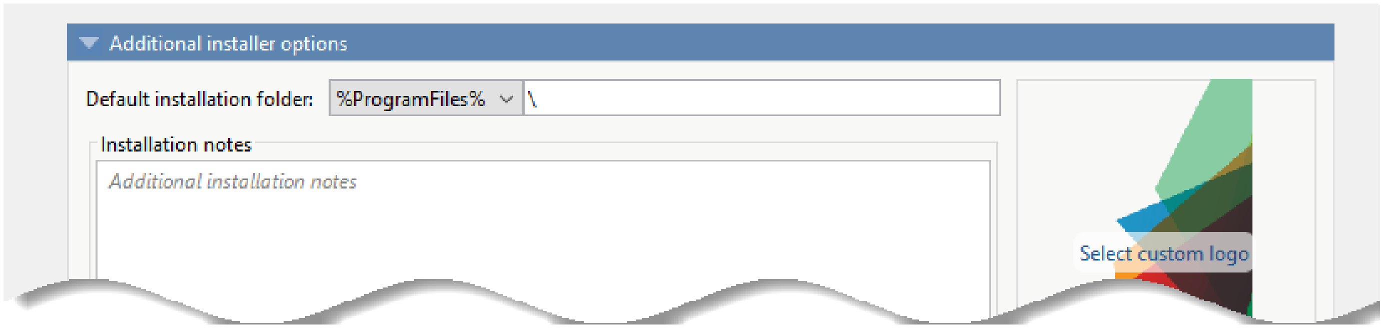
You can drag and drop a custom image onto the default splash screen.

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**.



Default installation folder: %ProgramFiles% \

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\`*`userName`*`\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.

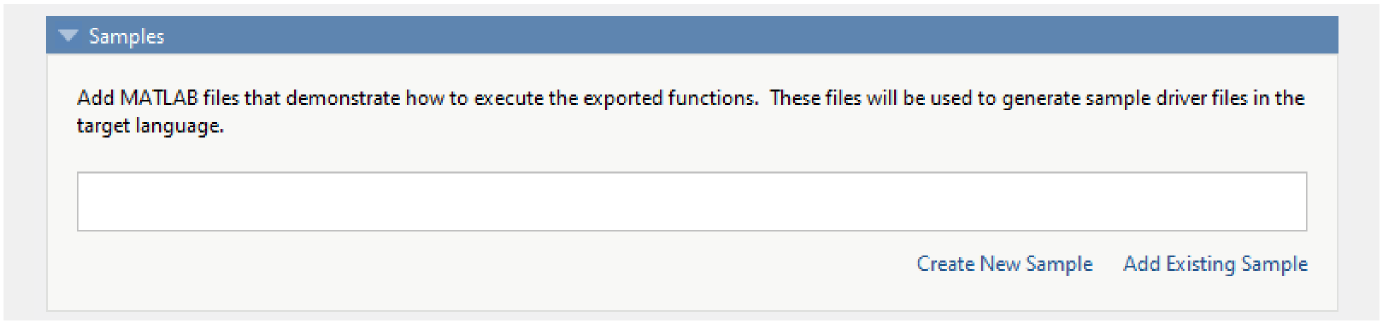
You can add files to the list by passing one or more `-a` arguments to `mcc`. The `-a` arguments add the specified files to the list of files to be added into the generated binary. For example, `-a hello.m` adds the file `hello.m` to the list of required files and `-a ./foo` adds all the files in `foo`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

Sample Driver File Creation

Sample driver files are used to implement the generated component into an application in the target language.

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

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



Add MATLAB files that demonstrate how to execute the exported functions. These files will be used to generate sample driver files in the target language.

The sample file creation feature in **Library Compiler** uses MATLAB code to generate sample files 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 file in the target language is generated from your MATLAB script and is saved in a folder named `samples`. Sample files are also included in the installer.

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 should edit it as necessary based on the behavior of your exported functions.

The sample MATLAB files must follow these guidelines:

- The sample file must be a MATLAB script, not a function.
- The sample file code must use only exported functions. Any user-defined function called in the script must be a top-level exported function.
- Each exported function must be in a separate sample file.
- Each call to the same exported function must be a separate sample file.
- The input parameters of the top-level function are analyzed during the process. An input parameter cannot be a field in a struct.
- 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.
- The sample script is executed as part of the process of generating the target language sample code. Any errors in execution (for instance, undefined variables) will prevent a sample from being generated, although the build target will still be generated.

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.

If you have already created a MATLAB sample file, you can include it in a `compiler.build` function for the supported targets using the `'SampleGenerationFiles'` option.

You can also choose not to include a sample file at all during the packaging step. If you create your own 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

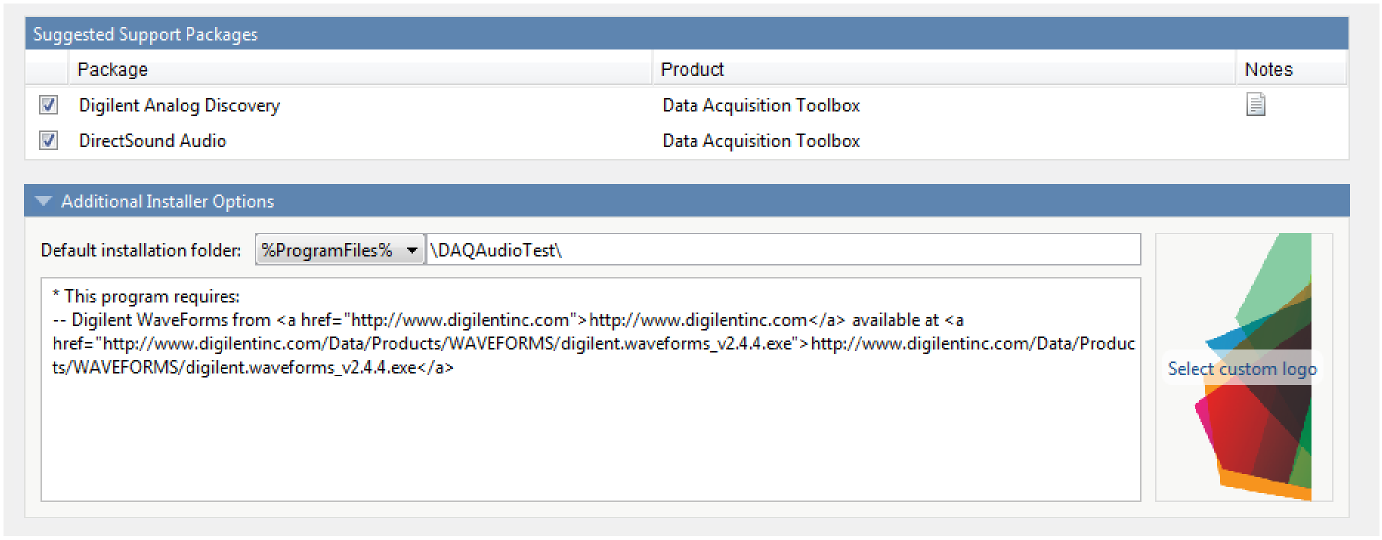
See Also

Application Compiler

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.



	Package	Product	Notes
✓	Digilent Analog Discovery	Data Acquisition Toolbox	📄
✓	DirectSound Audio	Data Acquisition Toolbox	

Default installation folder: %ProgramFiles% \DAQAudioTest\

```
* This program requires:
-- Digilent WaveForms from <a href="http://www.digilentinc.com">http://www.digilentinc.com</a> available at <a
href="http://www.digilentinc.com/Data/Products/WAVEFORMS/digilent.waveforms_v2.4.4.exe">http://www.digilentinc.com/Data/Produc
ts/WAVEFORMS/digilent.waveforms_v2.4.4.exe</a>
```

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\R2023a\toolbox\daq\supportpackages\daqaudio ...
-a 'C:\MATLAB\SupportPackages\R2023a\resources\daqaudio'
```

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 3-2
- “Invoke MATLAB Build Options” on page 3-4
- “MATLAB Runtime Component Cache and Deployable Archive Embedding” on page 3-6
- “mcc Command Arguments Listed Alphabetically” on page 3-8

Simplify Compilation Using Macros

In this section...
"Macros" on page 3-2
"Working With Macros" on page 3-2

Macros

The `mcc` function, through its exhaustive set of options, allows you to customize the behavior of a compiled component. If you want a simplified approach to compilation, you can use a *macro* 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

In this section...

"Specify Full Path Names to Build MATLAB Code" on page 3-4

"Using Bundles to Build MATLAB Code" on page 3-4

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
`cpplib`	C++ library	`-W cpplib:`*`library_name`* `-T link:lib`
`csharedlib`	C library	`-W lib:`*`library_name`* `-T link:lib`
`ccom`	COM component	`-W com:`*`component_name`*`,`*`className`*`,`*`version`* `-T link:lib`
`cexcel`	Excel Add-in	`-W excel:`*`addin_name`*`,`*`className`*`,`*`version`* `-T link:lib`
`cjava`	Java package	`-W java:`*`packageName`*`,`*`className`*
`dotnet`	.NET assembly	`-W dotnet:`*`assembly_name`*`,`*`className`*`,`*`framework_version`*`,`*`security`*`,`*`remote_type`* `-T link:lib`

MATLAB Runtime Component Cache and Deployable Archive Embedding

In this section...

"Overriding Default Behavior" on page 3-7
"For More Information" on page 3-7

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 `.ctf` 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 `.max_size`, which resides in the file designated by running the `mcrcachedir` 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 3-7 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 “About the Deployable Archive” (MATLAB Compiler).

mcc Command Arguments Listed Alphabetically

Option	Description	Comment
`-?`	Display `mcc` help message.	Cannot be used in a `deploytool` app.
`-a filepath`	Add `filepath` to the deployable archive.	If you specify a folder name, all files in the folder are added. If you use a wildcard (*), all files matching the wildcard are added.
`-A arch`	Append supported platforms to those detected automatically by the compiler.	Valid only for Python, C/C++ using the MATLAB data array API, and Java targets. *`arch`* = `win64`, `maci64`, `glnxa64`, or `all`
`-b`	Generate Excel® compatible formula function.	Requires MATLAB Compiler for Excel add-ins. Cannot be used in a `deploytool` app.
`-B bundle[:parameters]`	Replace `-B bundle` on the `mcc` command line with the contents of *`bundle`*.	The file should contain only `mcc` command-line options. MathWorks® included bundle files are located in *`matlabroot`*`\toolbox\compiler\bundles`.
`-c`	Suppress compiling and linking of the generated C wrapper code.	Must be used in conjunction with the `-l` option.
`-C`	Direct `mcc` to not embed the deployable archive in generated binaries.	
`-d outputfolder`	Place output in folder specified by *`outputfolder`*.	The specified folder must already exist. Cannot be used in a `deploytool` app.
`-e`	Suppresses appearance of the MS-DOS Command Window when generating a standalone application.	Use *`-e`* in place of the *`-m`* option. Available for Windows only. Equivalent to `-W WinMain -T link:exe`. Cannot be used in a `deploytool` app. The standalone app compiler suppresses the MS-DOS command window by default. To enable it, deselect **Do not display the Windows Command Shell (console) for execution** in the **Additional Runtime Settings** area.
`-f filename`	Use the specified options file, *`filename`*, when calling `mbuild`.	`mbuild -setup` is recommended. Valid for C/C++ shared libraries, COM, and Excel targets.
`-G`	Include debugging symbol information for generated C/C++ code.	
`-h helpfile`	Specify a custom help text file.	Display help file contents at runtime using `-?` or `/?`. Valid for standalone applications, C/C++ shared libraries, COM, and Excel targets.
`-I folder`	Add folder to search path for MATLAB files.	

Option	Description	Comment
`-j`	Automatically convert all `.m` files to P-files before packaging.	
`-k 'file=<keyfile>;loader=<mexfile>'`	Specify AES encryption key *`keyfile`* and MEX-file loader interface *`mexfile`* to retrieve decryption key at runtime.	If you do not specify any arguments after `-k`, `mcc` generates a 256-bit AES key and a loader MEX-file.
`-K`	Directs `mcc` to not delete output files if the compilation ends prematurely, due to error.	Default behavior is to dispose of any partial output if the command fails to execute successfully.
`-l`	Create a C shared library.	Equivalent to `-W lib -T link:lib`. Cannot be used in a `deploytool` app.
`-m`	Generate a standalone application.	Equivalent to `-W main -T link:exe`. Cannot be used in a `deploytool` app. On Windows, the command prompt opens on execution of the application. The standalone app compiler suppresses the MS-DOS command window by default. To enable it, deselect **Do not display the Windows Command Shell (console) for execution** in the **Additional Runtime Settings** area.
`-M options`	Pass compile-time options to `mbuild`.	
`-n`	Automatically treat numeric inputs as MATLAB doubles.	Cannot be used in a `deploytool` app.
`-N`	Clear the path of all but a minimal, required set of folders.	Uses the following folders: • *`matlabroot`*`\toolbox\matlab` • *`matlabroot`*`\toolbox\local` • *`matlabroot`*`\toolbox\compiler` • *`matlabroot`*`\toolbox\shared\bigdata`
`-o executablename`	Specify name of standalone application executable file.	Adds appropriate extension. Cannot be used in a `deploytool` app.
`-p folder`	Add *`folder`* to compilation path in an order-sensitive context.	Requires `-N` option.
`-r icon`	Embed resource *`icon`* in binary.	
`-R option`	Specify run-time options for MATLAB Runtime.	Valid only for standalone applications using MATLAB Compiler. *`option`* `=` `-nojvm`, `-nodisplay`, `'-logfile` *`filename`*`'`, `-startmsg`, and `-completemsg` *`filename`*

Option	Description	Comment
`-s`	Obfuscate folder structures and file names in the deployable archive (`.ctf` file) from the end user.	
`-S`	Create singleton MATLAB Runtime.	Default for generic COM components. Valid for Microsoft® Excel and Java packages.
`-T phase:type`	Specify the output target phase and type.	Cannot be used in a `deploytool` app.
`-u`	Registers COM component for current user only on development machine.	Valid only for generic COM components and Microsoft Excel add-ins.
`-U`	Generate a deployable archive (`.ctf` file) for MATLAB Production Server.	Equivalent to `-W 'CTF'`. Cannot be used in a `deploytool` app.
`-v`	Verbose; display compilation steps.	
`-w option[:warning]`	Control warning messages.	Valid arguments are `list`, `enable[:`*`warning`*`]`, `disable[:`*`warning`*`]`, `error[:`*`warning`*`]`, `on[:`*`warning`*`]`, and `off[:`*`warning`*`]`.
`-W 'target[:options]'`	Specify build target and associated options.	*`target`* = `main`, `WinMain`, `excel`, `hadoop`, `spark`, `lib`, `cpplib`, `com`, or `dotnet`, `java`, `python`, `CTF`, or `mpsxl`. Cannot be used in a `deploytool` app.
`-X`	Ignore data files detected by dependency analysis.	For more information, see “Dependency Analysis Using MATLAB Compiler” (MATLAB Compiler).
`-Y licensefile`	Override the default license file with the specified file *`licensefile`*.	Can only be used on the system command line.
`-Z supportpackage`	Specify method of including support packages.	*`supportpackage`* = `'autodetect'` (default), `'none'`, or *`packagename`*.

Packaging Log and Output Folders

By default, the deployment app places the packaging log and the **Testing Files**, **End User Files**, and **Packaged Installers** folders in the target folder location. If you specify a custom location, the app creates any folders that do not exist at compile time.

Functions

compiler.build.productionServerArchive

Create an archive for deployment to MATLAB Production Server or Docker

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 MATLAB 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:

- `includedSupportPackages.txt` — Text file that lists all support files included in the archive.
- `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.

For this example, use the files `addmatrix.m` and `subtractmatrix.mat` located in `*matlabroot*\extern\examples\compiler`.

```
addFile = fullfile(matlabroot,'extern','examples','compilersdk','c_cpp','matrix','addmatrix.m');
subFile = fullfile(matlabroot,'extern','examples','compilersdk','c_cpp','matrix','subtractmatrix.m');
```

Build a production server archive using the `compiler.build.productionServerArchive` command. Use name-value arguments to specify the archive name and enable verbose output.

```
compiler.build.productionServerArchive({addFile,subFile},...
    'ArchiveName','MatrixArchive',...
    'Verbose','on');
```

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

- `includedSupportPackages.txt` — Text file that lists all support files included in the archive.
- `MatrixArchive.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.

Create Multiple Production Server Archives Using Options Object

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

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

```
functionFile = fullfile(matlabroot,'extern','examples','compiler','hello.m');
```

Create a `ProductionServerArchiveOptions` object. Use name-value arguments to specify a common output directory, disable the automatic inclusion of data files, and enable verbose output.

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

```
opts = 

  ProductionServerArchiveOptions with properties:

            ArchiveName: 'hello'
          FunctionFiles: {'C:\Program Files\MATLAB\R2023a\extern\examples\compiler\hello.m'}
     FunctionSignatures: ''
        AdditionalFiles: {}
    AutoDetectDataFiles: off
        SupportPackages: {'autodetect'}
                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 `houdini.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 = 'houdini.m';
compiler.build.productionServerArchive(opts);
```

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

Create Microservice Docker Image Using Results

Create a microservice Docker® image using the results from building a production server archive on a Linux system.

Install and configure Docker on your system.

Create a production server archive using `magicsquare.m` and save the build results to a `compiler.build.Results` object.

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

Pass the `Results` object as an input to the `compiler.package.microserviceDockerImage` function to build the Docker image.

```
compiler.package.microserviceDockerImage(buildResults);
```

The function generates the following files within a folder named `magicsquaremicroserviceDockerImage` in your current working directory:

- `applicationFilesForMATLABCompiler/magicsquare.ctf` — Deployable archive file.
- `Dockerfile` — Docker file that specifies Docker run time options.
- `GettingStarted.txt` — Text file that contains deployment information.

For more details, see “Create Microservice Docker Image” (MATLAB Compiler SDK).

Get Build Information from Production Server Archive

Create a production server archive and save information about the build type, archive file, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

```
results = compiler.build.productionServerArchive(magicsquare.m')
```

```
results = 

  Results with properties:

                BuildType: 'productionServerArchive'
                    Files: {'D:\Documents\MATLAB\work\magicsquareproductionServerArchive\magicsquare.ctf'}
    IncludedSupportPackages: {}
                  Options: [1×1 compiler.build.ProductionServerArchiveOptions]
```

The `Files` property contains the path to the deployable archive file `magicsquare.ctf`.

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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\signatures.json'`

Data Types: `char` | `string`

`ObfuscateArchive` — Flag to obfuscate deployable archive
`'off'` (default) | on/off logical value

Flag to obfuscate the deployable archive, 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 folder structures and file names in the deployable archive are obfuscated from the end user, and user code and data contained in MATLAB files are placed into a user package within the archive. Additionally, all `.m` files are converted to P-files before packaging. This option is equivalent to using `mcc` with `-j` and `-s` specified.
- If you set this property to `'off'`, then the deployable archive is not obfuscated. This is the default behavior.

Example: `'ObfuscateArchive','on'`

Data Types: `logical`

`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'`

`SupportPackages` — Support packages
`'autodetect'` (default) | `'none'` | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- `'autodetect'` (default) — The dependency analysis process detects and includes the required support packages automatically.
- `'none'` — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`

Example: `'SupportPackages',{'Deep Learning Toolbox Converter for TensorFlow Models','Deep Learning Toolbox Model for Places365-GoogLeNet Network'}`

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

`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:

- The build type, which is `'productionServerArchive'`
- Path to the deployable archive file
- A list of included support packages
- Build options, specified as a `ProductionServerArchiveOptions` object

Version History

Introduced in R2020b

See Also

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

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\R2023a\extern\examples\compiler\magicsquare.m'}
     FunctionSignatures: ''
        AdditionalFiles: {}s+ AutoDetectDataFiles: ons+ ObfuscateArchive: offs+ SupportPackages: {'autodetect'}
              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\R2023a\extern\examples\compiler\magicsquare.m'}
```

```
     FunctionSignatures: ''
        AdditionalFiles: {}s+ AutoDetectDataFiles: ons+ ObfuscateArchive: offs+ SupportPackages: {'autodetect'}
              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: {2×1 cell}
     FunctionSignatures: ''
        AdditionalFiles: {}
    AutoDetectDataFiles: off
        SupportPackages: {'autodetect'}
              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: {2×1 cell}
     FunctionSignatures: ''
        AdditionalFiles: {}
    AutoDetectDataFiles: off
        SupportPackages: {'autodetect'}
              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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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\signatures.json'`

Data Types: `char` | `string`

`ObfuscateArchive` — Flag to obfuscate deployable archive
`'off'` (default) | on/off logical value

Flag to obfuscate the deployable archive, 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 folder structures and file names in the deployable archive are obfuscated from the end user, and user code and data contained in MATLAB files are placed into a user package within the archive. Additionally, all `.m` files are converted to P-files before packaging. This option is equivalent to using `mcc` with `-j` and `-s` specified.
- If you set this property to `'off'`, then the deployable archive is not obfuscated. This is the default behavior.

Example: `'ObfuscateArchive','on'`

Data Types: `logical`

`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'`

`SupportPackages` — Support packages
`'autodetect'` (default) | `'none'` | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- `'autodetect'` (default) — The dependency analysis process detects and includes the required support packages automatically.
- `'none'` — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

Example: `'SupportPackages',{'Deep Learning Toolbox Converter for TensorFlow Models','Deep Learning Toolbox Model for Places365-GoogLeNet Network'}`

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

`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.

Version History

Introduced in R2020b

See Also

`compiler.build.productionServerArchive`

compiler.build.Results

Compiler build results object

Description

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

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

For information on results from compiling standalone applications, Excel add-ins, or web app archives, see `compiler.build.Results` for MATLAB Compiler.

For information on results from compiling C/C++ shared libraries, .NET assemblies, COM components, Java packages, Python packages, MATLAB Production Server deployable archives, or Excel add-ins for MATLAB Production Server, see `compiler.build.Results` for MATLAB Compiler SDK.

Creation

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

- Create a production server archive using `compiler.build.productionServerArchive` (example (MATLAB Compiler SDK)).
- Create an Excel add-in for MATLAB Production Server using `compiler.build.excelClientForProductionServer` (example (MATLAB Compiler SDK)).

Properties

`BuildType` — Build type
`'productionServerArchive'` | `'excelClientForProductionServer'`

This property is read-only.

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

`compiler.build` Function	Build Type
`compiler.build.excelClientForProductionServer`	`'excelClientForProductionServer'`

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\MagicSquareproductionServerArchive\MagicSquare.ctf'}`

Data Types: `cell`

`IncludedSupportPackages` — Support packages
cell array of character vectors

This property is read-only.

Support packages included in the generated component, specified as a cell array of character vectors.

`Options` — Build options
`ProductionServerArchiveOptions` | `ExcelClientForProductionServerOptions`

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.

Build Type	Options

Examples

Get Build Information from Production Server Archive

Create a production server archive and save information about the build type, archive file, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

```
results = compiler.build.productionServerArchive(magicsquare.m')
```

```
results = 

  Results with properties:

                BuildType: 'productionServerArchive'
                    Files: {'D:\Documents\MATLAB\work\magicsquareproductionServerArchive\magicsquare.ctf'}
    IncludedSupportPackages: {}
                  Options: [1×1 compiler.build.ProductionServerArchiveOptions]
```

The `Files` property contains the path to the deployable archive file `magicsquare.ctf`.

Get Build Information from Excel Add-In for MATLAB Production Server

Create an Excel add-in for MATLAB Production Server and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Build a MATLAB Production Server archive using the file `magicsquare.m`. Save the output as a `compiler.build.Results` object `serverBuildResults`.

```
serverBuildResults = compiler.build.productionServerArchive('magicsquare.m');
```

Build the Excel add-in using the `serverBuildResults` object.

```
results = compiler.build.excelClientForProductionServer(serverBuildResults)
```

```
results = 

  Results with properties:

            BuildType: 'excelClientForProductionServer'
                Files: {1×1 cell}
IncludedSupportPackages: {}
              Options: [1×1 compiler.build.ExcelClientForProductionServerOptions]
```

The `Files` property contains the paths to the following compiled files:

- `magicsquare.dll`
- `magicsquare.bas`
- `magicsquare.xla`

Note The files `magicsquare.bas` and `magicsquare.xla` are included in `Files` only if you enable the `'GenerateVisualBasicFile'` option in the `compiler.build.excelClientForProductionServer` command.

Version History

Introduced in R2020b

See Also

`compiler.build.productionServerArchive` | `compiler.build.excelClientForProductionServer`

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.

Version History

Introduced in R2014a

R2020a: `-build` and `-package` options will be removed
Warns 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.

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
```

A list opens with the following options:

- **Application Compiler**
- **Hadoop Compiler**
- **Library Compiler**
- **Production Server Compiler** (if MATLAB Compiler SDK is installed)
- **Web App Compiler**

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.

Version History

R2020a: `-build` and `-package` options will be removed
Warns 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(options,mfilename)

mcc -m [options] mfilename
mcc -e [options] mfilename

mcc -W 'excel:addin_name,class_name,version=version_number' [options]
mfilename1 mfilename2 ... mfilenameN

mcc -W 'hadoop:archive_name,CONFIG:config_file' mfilename

mcc -m [options] mfilename

mcc -W python:package_name [options] mfilename1 mfilename2 ... mfilenameN

mcc -W
'dotnet:assembly_name,api=api_type,class_name,framework_version,security,remo
te_type' [options] mfilename1 mfilename2 ... mfilenameN
mcc -W
'dotnet:assembly_name,api=api_type,class_name,framework_version,security,remo
te_type' [options] 'class{class_name:mfilename1,mfilename2,...,mfilenameN}'

mcc -W 'java:package_name,class_name' [options] mfilename1 mfilename2 ...
mfilenameN
mcc -W 'java:package_name,class_name' [options] '
class{class_name:mfilename1,mfilename2,...,mfilenameN}'

mcc -l [options] mfilename1 mfilename2 ... mfilenameN

mcc -W 'cpplib:library_name[,{all|legacy|generic}]' [options] mfilename1
mfilename2 ... mfilenameN

mcc -W 'com:component_name,class_name' [options] mfilename1 mfilename2 ...
mfilenameN
mcc -W 'com:component_name,class_name' [options] '
class{class_name:mfilename1,mfilename2,...,mfilenameN}'

mcc -U -W 'CTF:archive_name,DISCOVERY:FunctionSignatures.json' [options]
mfilename1 mfilename2 ... mfilenameN

mcc -W 'mpsxl:addin_name,class_name,version' input_marshaling_flags
output_marshaling_flags [options] mfilename1 mfilename2 ... mfilenameN
```

Description

You can use `mcc` to package and deploy MATLAB programs as standalone applications, Excel add-ins, Spark™ applications, or Hadoop® jobs.

If you have a MATLAB Compiler SDK license, you can use `mcc` to create C/C++ shared libraries, .NET assemblies, Java packages, Python packages, MATLAB Production Server deployable archives, or Excel add-ins for MATLAB Production Server.

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. The first file acts as the entry point to the compiled artifact.

You can also call this syntax from a system command prompt.

Note Arguments that contain special characters (such as period or space) must be surrounded by single quotes. Use double quotes when executing from a Windows command prompt.

`mcc(options,mfilename)` compiles the function as specified by the options. Specify file names and options as character vectors or strings. This syntax allows you to use MATLAB variables as input arguments.

Standalone Application

`mcc -m [options] mfilename` compiles the function into a standalone application. The executable type is determined by your operating system.

As an alternative, the `compiler.build.standaloneApplication` function supports most common workflows.

`mcc -e [options] mfilename` compiles the function into a standalone application that does not open a Windows command prompt on execution. The `-e` option works only on Windows operating systems.

As an alternative, the `compiler.build.standaloneWindowsApplication` function supports most common workflows.

Excel Add-In

`mcc -W 'excel:`*`addin_name`*`,`*`class_name`*`,version=`*`version_number`*`' [options] mfilename1 mfilename2 ... mfilenameN` creates a Microsoft Excel add-in using the specified files. Before creating Excel add-ins, install a supported compiler.

You can only create Excel add-ins on Windows.

- *`addin_name`* — Specifies the name of the add-in. If you do not specify the name, `mcc` uses *`mfilename1`* as the default.
- *`class_name`* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses *`addin_name`* as the class name. If specified, *`class_name`* must be different from *`mfilename1`*.

- `version_number` — Specifies the version number of the add-in file as `major.minor.bug.build` in the file system. You are not required to specify a version number. If you do not specify a version number, `mcc` sets the version number to `1.0.0.0` by default.
  - `major` — Specifies the major version number. If you do not specify a number, `mcc` sets `major` to `1`.
  - `minor` — Specifies the minor version number. If you do not specify a number, `mcc` sets `minor` to `0`.
  - `bug`— Specifies the bug fix maintenance release number. If you do not specify a number, `mcc` sets `bug` to `0`.
  - `build`— Specifies the build number. If you do not specify a number, `mcc` sets `build` to `0`.

As an alternative, the `compiler.build.excelAddIn` function supports most common workflows.

MapReduce Applications on Hadoop

Linux only

`mcc -W 'hadoop:archive_name,CONFIG:config_file' mfilename` generates a deployable archive from `mfilename` that can be run as a job by Hadoop.

- `archive_name` — Specifies the name of the generated archive.
- `config_file` — Specifies the path to the configuration file for creating a deployable archive. For more information, see “Configuration File for Creating Deployable Archive Using the mcc Command” (MATLAB Compiler).

Simulink Simulations

Requires Simulink Compiler

`mcc -m [options] mfilename` compiles a MATLAB application that contains a Simulink simulation into a standalone application. For more information, see “Create and Deploy a Script with Simulink Compiler” (Simulink Compiler).

Python Package

Requires MATLAB Compiler SDK

`mcc -W python:package_name [options] mfilename1 mfilename2 ... mfilenameN` creates a Python package using the specified files.

- `package_name` — Specifies the name of the Python package preceded by an optional namespace, which is a period-separated list such as `companyname.groupname.component`.

As an alternative, the `compiler.build.pythonPackage` function supports most common workflows.

.NET Assembly

`mcc -W 'dotnet:assembly_name,api=api_type,class_name,framework_version,security,remote_type' [options] mfilename1 mfilename2 ... mfilenameN` creates a .NET assembly with a single class using the specified files. Before creating .NET assemblies, see “MATLAB Compiler SDK .NET Target Requirements” (MATLAB Compiler SDK).

- `assembly_name` — Specifies the name of the assembly preceded by an optional namespace, which is a period-separated list such as `companyname.groupname.component`.
- `api_type` — Specifies the API type of the assembly. Values are `matlab-data` and `mwarray`. The default value is `mwarray`.
- `class_name` — Specifies the name of the .NET class to be created.
- `framework_version` — Specifies the version of the Microsoft .NET Framework you want to use to compile the assembly. Specify either:

  - `0.0` — Use the latest supported version on the target machine.
  - `version_major.version_minor` — Use a specific version of the framework.

  Features are often version-specific. Consult the documentation for the feature you are implementing to get the Microsoft .NET Framework version requirements.
- `security` — Specifies whether the assembly to be created is a private assembly or a shared assembly.

  - To create a private assembly, specify `Private`.
  - To create a shared assembly, specify the full path to the encryption key file used to sign the assembly.
- `remote_type` — Specifies the remoting type of the assembly. Values are `remote` and `local`.

`mcc -W 'dotnet:assembly_name,api=api_type,class_name,framework_version,security,remote_type' [options] 'class{class_name:mfilename1,mfilename2,...,mfilenameN}'` creates a .NET assembly with multiple classes using the specified files. You can include additional class specifiers by adding `class{___}` arguments.

As an alternative, the `compiler.build.dotNETAssembly` function supports most common workflows.

Java Package

Requires MATLAB Compiler SDK

`mcc -W 'java:package_name,class_name' [options] mfilename1 mfilename2 ... mfilenameN` creates a Java package from the specified files. Before creating Java packages, see Configure Your Java Environment (MATLAB Compiler SDK).

- `package_name` — Specifies the name of the Java package preceded by an optional namespace, which is a period-separated list such as `companyname.groupname.component`.
- `class_name` — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the last item in `package_name`.

`mcc -W 'java:package_name,class_name' [options] 'class{class_name:mfilename1,mfilename2,...,mfilenameN}'` creates a Java package with multiple classes from the specified files. You can include additional class specifiers by adding `class{___}` arguments.

As an alternative, the `compiler.build.javaPackage` function supports most common workflows.

C Shared Library

Requires MATLAB Compiler SDK

`mcc -l [options] mfilename1 mfilename2 ... mfilenameN` compiles the listed functions into a C shared library and generates C wrapper code for integration with other applications.

As an alternative, the `compiler.build.cSharedLibrary` function supports most common workflows.

C++ Shared Library

Requires MATLAB Compiler SDK

`mcc -W 'cpplib:`*`library_name`*`[,{all|legacy|generic}]' [options] mfilename1 mfilename2 ... mfilenameN` compiles the listed functions into a C++ shared library and generates C++ wrapper code for integration with other applications.

- *`library_name`* — Specifies the name of the shared library.
- `all`— Generates shared libraries using both the `mwArray` API and the generic interface that uses the MATLAB Data API. This is the default behavior.
- `legacy`— Generates shared libraries using the `mwArray` API.
- `generic`— Generates shared libraries using the MATLAB Data API.

As an alternative, the `compiler.build.cppSharedLibrary` function supports most common workflows.

COM Component

Requires MATLAB Compiler SDK

`mcc -W 'com:`*`component_name`*`,`*`class_name`*`' [options] mfilename1 mfilename2 ... mfilenameN` compiles the listed functions into a generic Microsoft COM component.

- *`component_name`* — Specifies the name of the COM component.
- *`class_name`* — Specifies the name of the class.

As an alternative, the `compiler.build.comComponent` function supports most common workflows.

`mcc -W 'com:component_name,`*`class_name`*`' [options] ' class{class_name:mfilename1,mfilename2,...,mfilenameN}'` creates a Microsoft COM component with multiple classes from the specified files. You can include additional class specifiers by adding `class{___}` arguments.

Deployable Archive for MATLAB Production Server

Requires MATLAB Compiler SDK

`mcc -U -W 'CTF:`*`archive_name`*`,DISCOVERY:`*`FunctionSignatures.json`*`' [options] mfilename1 mfilename2 ... mfilenameN` creates a deployable archive (`.ctf` file) for use with a MATLAB Production Server instance.

- *`archive_name`* — Specifies the name of the deployable archive.
- *`FunctionSignatures.json`* — JSON file that contains information about your MATLAB functions. This is only relevant for RESTful clients using the discovery API. For more information, see “MATLAB Function Signatures in JSON”.

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

As an alternative, the `compiler.build.productionServerArchive` function supports most common workflows.

Excel Add-In for MATLAB Production Server

Requires MATLAB Compiler SDK

`mcc -W 'mpsxl:`*`addin_name`*`,`*`class_name`*`,`*`version`*`'` *`input_marshaling_flags`* *`output_marshaling_flags`* `[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.
- *`class_name`* — 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 marshalled 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.

As an alternative, the `compiler.build.excelClientForProductionServer` function supports most common workflows.

Examples

Create Standalone Application

Create a standalone application and include the data file `data.mat`.

In MATLAB, locate the MATLAB code that you want to deploy as a standalone application. For this example, compile using the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

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

Build a standalone application with a reference to `appFile` using the MATLAB command syntax. Use this command at the MATLAB command prompt.

```
mcc('-m',appFile,'-a','data.mat','-v')
```

The function generates a standalone application named `magicsquare`. The executable file type depends on the operating system on which the application is created.

As an alternative, the `compiler.build.standaloneApplication` function supports most common workflows.

Create Standalone Application with No Command Prompt (Windows only)

Create a standalone application on Windows that does not open a command prompt window on execution.

You can use the `mcc` command at the MATLAB command prompt or the Windows command window.

```
mcc -e myapp.mlapp -o VisualApp
```

The function generates a standalone Windows application named `VisualApp`.

As an alternative, the `compiler.build.standaloneWindowsApplication` function supports most common workflows.

Create Excel Add-In (Windows only)

Create an Excel add-in on Windows with the system level version number `5.2.1.7`.

To generate the Excel add-in file (`.xla`), you must enable "Trust access to the VBA project object model" in your Excel settings. If you do not do this, you can manually create the add-in by importing the generated `.bas` file into Excel.

```
mcc -W 'excel:magicExcel,myClass,version=5.2.1.7' -b mymagic.m
```

The function generates an Excel add-in named `magicExcel`.

As an alternative, the `compiler.build.excelAddIn` function supports most common workflows.

Create Hadoop MapReduce Application

Create a MapReduce application on a Linux system that can be run as a job by Hadoop.

Create a configuration file named `config.txt` in your work folder that specifies configuration info.

```
mw.ds.in.type = tabulartext
mw.ds.in.format = infoAboutDataset.mat
mw.ds.out.type = keyvalue
mw.mapper = maxArrivalDelayMapper
mw.reducer = maxArrivalDelayReducer
```

For more information, see "Configuration File for Creating Deployable Archive Using the mcc Command" (MATLAB Compiler).

Copy the example files `maxArrivalDelayMapper.m`, `maxArrivalDelayReducer.m`, and `airlinesmall.csv` located in the folder *`matlabroot`*`/toolbox/matlab/demos` to your work folder.

Compile the files using the `mcc` command.

```
mcc -W 'hadoop:maxArrivalDelay,CONFIG:config.txt' maxArrivalDelayMapper.m maxArrivalDelayReducer
```

The function generates a shell script named `run_maxarrivaldelay.sh`, a deployable archive named `maxArrivalDelayMapper.ctf`, and a readme file with usage details.

For more details, see "Include MATLAB Map and Reduce Functions into Hadoop Job" (MATLAB Compiler).

Create Simulink Simulation

Create a standalone application that runs a Simulink Simulation.

Create a Simulink model using Simulink. This example uses the model `sldemo_suspn_3dof`.

Create a MATLAB application that uses APIs from Simulink Compiler to simulate the model. For more information, see "Deploy Simulations with Tunable Parameters" (Simulink Compiler).

```
function deployParameterTuning(outputFile, mbVariable)

    if ischar(mbVariable) || isstring(mbVariable)
        mbVariable = str2double(mbVariable);
    end

    if isnan(mbVariable) || ~isa(mbVariable, 'double') || ~isscalar(mbVariable)
        disp('mb must be a double scalar or a string or char that can be converted to a double scalar');
    end

    in = Simulink.SimulationInput('sldemo_suspn_3dof');
    in = in.setVariable('Mb', mbVariable);
    in = simulink.compiler.configureForDeployment(in);
    out = sim(in);

    save(outputFile, 'out');

end
```

Use `mcc` to create a standalone application from the MATLAB application.

```
mcc -m deployParameterTuning.m
```

The function generates an executable named `deployParameterTuning`.

For more information, see "Create and Deploy a Script with Simulink Compiler" (Simulink Compiler).

Create Python Package

Requires MATLAB Compiler SDK

Create a Python package using the file `mymagic.m` located in the subfolder `pymagic`.

```
mcc -W python:magicPython pymagic/mymagic.m
```

The function generates a Python package named `magicPython`.

As an alternative, the `compiler.build.pythonPackage` function supports most common workflows.

Create .NET Assembly Using MATLAB Data API

Requires MATLAB Compiler SDK

Create a MATLAB Data API .NET assembly using the file `mymagic.m`. Specify a namespace, class name, framework version, security, and remoting type in the `-W` argument.

```
mcc -W 'dotnet:company.group.magic,api=matlab-data,dotnetClass,0.0,Private,local' mymagic.m
```

The function generates a .NET assembly archive named `magic.ctf`.

As an alternative, the `compiler.build.dotNETAssembly` function supports most common workflows.

Create .NET Assembly Using `mwArray`

Requires MATLAB Compiler SDK

Create a `mwArray` .NET assembly using the file `mymagic.m`. Specify a class name and framework version using the `-W` argument.

```
 mcc -W 'dotnet:magic,magicClass,5.0' mymagic.m
```

The function generates a .NET assembly named `magic.dll`.

As an alternative, the `compiler.build.dotNETAssembly` function supports most common workflows.

Create Java Package

Requires MATLAB Compiler SDK

Create a Java package using the file `mymagic.m`. Specify a class name and namespace using the `-W` argument.

```
mcc -W 'java:company.group.javamagic,magicClass' mymagic.m
```

The function generates a Java package named `magic.jar`.

As an alternative, the `compiler.build.javaPackage` function supports most common workflows.

Create C Shared Library

Requires MATLAB Compiler SDK

Create a C shared library using the file `mymagic.m`.

```
mcc -W lib:magiclibrary mymagic.m
```

The function generates a C shared library named `magic.dll`.

As an alternative, the `compiler.build.cSharedLibrary` function supports most common workflows.

Create C++ Shared Library Using MATLAB Data API

Requires MATLAB Compiler SDK

Create a C++ shared library that uses the MATLAB Data API using the file `mda_magic.m`.

```
mcc -W 'cpplib:matlabmagiccpp,generic' mda_magic.m
```

The function generates a C++ shared library archive named `matlabmagiccpp.ctf`.

As an alternative, the `compiler.build.cppSharedLibrary` function supports most common workflows.

Create C++ Shared Library Using `mwArray`

Requires MATLAB Compiler SDK

Create a C++ shared library that uses the `mwArray` API using the file `mwa_magic.m`.

```
mcc -W 'cpplib:mwarraymagiccpp,legacy' mwa_magic.m
```

The function generates a C++ shared library named `mwarraymagiccpp.dll`.

As an alternative, the `compiler.build.cppSharedLibrary` function supports most common workflows.

Create COM Component

Requires MATLAB Compiler SDK

Create a COM component using the files `mymagic.m`, `data2.m`, and `data2.m`. Use the `class` argument to map the magic function to a class named `MagicClass` and the data functions to a class named `DataClass`.

```
mcc -W com:magicCOM 'class{MagicClass:mymagic.m}' 'class{DataClass:data1.m,data2.m}'
```

The function generates a COM component named `magicCOM_1_0.dll`.

As an alternative, the `compiler.build.comComponent` function supports most common workflows.

Create Deployable Archive for MATLAB Production Server

Requires MATLAB Compiler SDK

Create a MATLAB Production Server archive using the file `mymagic.m`. Specify a function signatures JSON file using the `-W` argument.

```
mcc -U -W 'CTF:mps_magic,DISCOVERY:magicFunctionSignatures.json' mymagic.m
```

The function generates a deployable archive named `mps_magic.ctf`.

As an alternative, the `compiler.build.productionServerArchive` function supports most common workflows.

Create Excel add-in for MATLAB Production Server

Requires MATLAB Compiler SDK

Create a MATLAB Production Server archive using the file `mymagic.m`.

```
mcc -U -W CTF:mps_magic mymagic.m
```

Next, create an Excel add-in for MATLAB Production Server.

To generate the Visual Basic files, enable **Trust access to the VBA project object model** in Excel. If you do not do this, you can manually create the add-in by importing the `.bas` file into Excel.

```
mcc -W 'mpsxl:magicAddin,myExcelClass,version=1.0' mymagic.m
```

The function generates an Excel add-in named `magicAddin`.

As an alternative, the `compiler.build.excelClientForProductionServer` function supports most common workflows.

Input Arguments

Tip To view a table of `mcc` input arguments in alphabetical order, see “mcc Command Arguments Listed Alphabetically” (MATLAB Compiler).

`mfilename` — File to be compiled
file name

File to be compiled, specified as a character vector or string scalar.

`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. The first file is used as the entry point for the compiled artifact.

`class{`*`class_name`*`:mfilename1,mfilename2,...,mfilenameN}` — Files to be included in a class
list of file names

One or more files to be included in the class *`class_name`*, specified as a comma-separated list of file names. You can include multiple class specifiers by adding additional `class{___}` arguments. The argument applies only to the COM component, Java package, and .NET assembly targets.

Target and Platform

`-W 'target:artifact_name[,options]'` — Build target
`main` | `WinMain` | `excel` | `hadoop` | `spark` | `lib` | `cpplib` | `com` | `dotnet` | `java` | `python` | `CTF` | `mpsxl`

Build target and associated options, specified as one of the syntaxes listed below.

The compiler generates wrapper functions that allow another programming language to run the corresponding MATLAB function and any necessary global variable definitions. You cannot use this option in a `deploytool` app.

Target	Syntax	Equivalent Option
Standalone Application	`-W 'main:`*`app_name`*`,version=`*`version`*`'`	`-m`
Standalone Application (no Windows console)	`-W 'WinMain:`*`app_name`*`,version=`*`version`*`'`	`-e`
Excel Add-In	`-W 'excel:`*`addin_name`*`,`*`class_name`*`,version=`*`version`*`'`	None
Hadoop MapReduce Application	`-W 'hadoop:`*`archive_name`*`,CONFIG:`*`configFile`*`'`	None
Spark Application	`-W 'spark:`*`app_name`*`,`*`version`*`'`	None

The following targets require MATLAB Compiler SDK.

Target	Syntax	Equivalent Option
C Shared Library	`-W 'lib:`*`library_name`*`'`	`-l`
C++ Shared Library	`-W 'cpplib:`*`library_name`*`[,{all|legacy|generic}]'`	None
COM Component	`-W 'com:`*`component_name`*`,`*`class_name`*`'`	None

Target	Syntax	Equivalent Option
.NET Assembly	`-W 'dotnet:`*`assembly_name`*`,api={matlab-data|mwarray},`*`class_name`*`,`*`framework_version`*`,`*`security`*`,{remote|local}'`	None
Java Package	`-W 'java:`*`package_name`*`,`*`class_name`*`'`	None
Python Package	`-W 'python:`*`package_name`*`,`*`class_name`*`'`	None
MATLAB Production Server Deployable Archive	`-W 'CTF:`*`archive_name`*`'`	None
MATLAB Production Server Excel Add-In	`-W 'mpsxl:`*`addin_name`*`,`*`class_name`*`,`*`version`*`'`	None

Note `-W` values that contain special characters, such as commas or periods, must be surrounded by single quotes. Use double quotes when executing from a Windows command prompt.

`-T phase:type` — Output target phase and type

`compile:exe` | `compile:lib` | `link:exe` | `link:lib`

Output target phase and type, specified as one of the following options. If not specified, `mcc` uses the default type for the target specified by the `-W` option.

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.

Example: `-T link:lib`

`-A arch` — Add platform

`win64` | `maci64` | `glnxa64` | `all`

Add the platform designated by *`arch`* to the list of compatible platforms detected automatically by the compiler. Valid platforms are `win64`, `maci64`, `glnxa64`, and `all`. The `-A` option only applies to the Python, Java, and C++ MATLAB Data API targets.

Running the component on an incompatible platform will result in an unsupported platform error message that lists compatible platforms.

`-B bundle[:parameters]` — Options bundle file
`ccom` | `cexcel` | `cjava` | `cmpsxl` | `cpplib` | `csharedlib` | `dotnet` | file name

Specify an options bundle file, where *`bundle`* is the name of a file that contains a set of `mcc` command line options, arguments, filenames, and/or other -B options. MathWorks included bundle files are located in *`matlabroot`*`\toolbox\compiler\bundles`.

A bundle can include replacement parameters for compiler options that accept names and version numbers. If more than one parameter is passed, you must enclose the expression that follows the `-B` in single quotes. For example, `mcc -B 'cexcel:component,class,1.0' ...`.

In general, each `%n%` in the bundle will be replaced with the corresponding option specified to the bundle. Use `%%` to include a literal `%` character. It is an error to pass too many or too few options to the bundle. For more details, see "Using Bundles to Build MATLAB Code" (MATLAB Compiler SDK).

Available Bundle Files

Bundle File	Target	Contents
`ccom`	COM component	`-W com:%1%,%2%,%3% -T link:lib`
`cexcel`	Excel Add-in	`-W excel:%1%,%2%,%3% -T link:lib -b -S`
`cjava`	Java package	`-W java:%1%,%2%`
`cmpsxl`	Excel Add-In for MATLAB Production Server	`-W mpsxl:%1%,%2%,%3% -T link:lib`
`cpplib`	C++ library	`-W cpplib:%1% -T link:lib`
`csharedlib`	C library	`-W lib:%1% -T link:lib`
`dotnet`	.NET assembly	`-W dotnet:%1%,%2%,%3%,%4%,%5% -T link:lib`

Standalone Applications

`-m` — Generate standalone application

Generate a standalone application. `-m` is equivalent to `-W main -T link:exe`. On Windows, the command prompt opens on execution of the application.

You cannot use this option in a `deploytool` app.

`-e` — Generate standalone Windows application

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. You cannot use this option in a `deploytool` app.

`-o executablename` — Executable name
file name

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

Example: `-o myexecutable`

`-r icon` — Add icon resource

file path

Add icon resource to the executable of a standalone application. Paths can be relative to the current working directory or absolute.

Example: `-r path\to\icon.ico`

`-n` — Interpret command line inputs as MATLAB doubles

Interpret command line inputs as MATLAB doubles. If you do not specify this option, command line inputs are treated as MATLAB character vectors.

Excel Add-Ins and COM Components

`-b` — Generate Visual Basic file

Generate a Visual Basic file (`.bas`) and an Excel add-in file (`.xla`). The `.bas` file contains the Microsoft Excel Formula Function interface to the COM object generated by MATLAB Compiler. When imported into a workbook, this Visual Basic code allows the MATLAB function to be used as a cell formula function.

Note To generate the Excel add-in file (`.xla`), you must enable "Trust access to the VBA project object model" in your Excel settings.

`-u` — Register COM component for current user

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.

C Shared Libraries

`-l` — Generate C shared library

Generate a C shared library. `-l` is equivalent to `-W lib -T link:lib`. You cannot use this option in a `deploytool` app.

`-c` — Suppress code linking

Suppress compiling and linking the generated C wrapper code. The `-c` option cannot be used independently of the `-l` option.

MATLAB Production Server

`-U` — Generate MATLAB Production Server archive

Generate a MATLAB Production Server archive (`.ctf` file). This argument must be before `mfilename`, and you must also specify the option `-W 'CTF:`*`archive_name`*`'`.

Additional Files

-a filepath — Add file or folder
file path

Add file or folder to the deployable archive. File paths can be relative or absolute. For additional details, see "Access Files in Packaged Applications" (MATLAB Compiler).

To add multiple files, use multiple `-a` options, specify a folder, or use wildcards.

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 file name includes the 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, the following command adds all files in `./testdir` to the deployable archive, and subfolders under `./testdir` are not processed recursively.

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

The following command adds all files with the extension `.m` in `./testdir`, and subfolders of `./testdir` are not processed recursively.

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

Including Java Classes

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.

-h helpfile — Add help text file
file path

Add a custom help text file. Paths can be relative to the current working directory or absolute. This option applies to standalone applications, C/C++ shared libraries, COM, and Excel targets.

Display help file contents by calling the application at the command line with the `-?` or `/?` argument.

Example: `-h path\to\helpfile`

-X — Exclude data files

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

-Z supportpackage — Specify support packages
`autodetect` | `none` | Support package

Specify the method of adding support packages to the deployable archive as one of the following options.

Syntax	Description
`-Z autodetect`	The dependency analysis process detects and includes the required support packages automatically. This is the default behavior of mcc.
`-Z none`	No support packages are included. Using this option can cause runtime errors.
`-Z '`*`packagename`*`'`	Only the specified support package is included. To specify multiple support packages, use multiple `-Z` inputs.

Example: `-Z 'Deep Learning Toolbox Converter for TensorFlow Models'`

mcc Build Options

`-d outputfolder` — Output folder

folder name

Place build output in the specified folder *`outputfolder`*. Paths can be relative to the current directory or absolute.

`-C` — Do not embed deployable archive

Do not embed the deployable archive in binaries. This option is ignored for Java libraries.

`-Y licensefile` — License file

file name

Override the default license file with the specified file *`licensefile`*. This option can only be used on the system command line.

`-?` — Display help text

Display `mcc` help text in the console. This option cannot be used in a `deploytool` app.

Protect Source Code

`-j` — Obfuscate `.m` files

Obfuscate `.m` files. This option generates a P-code file with a `.p` extension for each `.m` file included in the `mcc` command before packaging.

P-code files are an obfuscated, execute-only form of MATLAB code. For more details, see `pcode`.

`-k 'file=<keyfile>;loader=<mexfile>'` — Specify encryption key and loader interface

key file and MEX-file

Specify an AES encryption key and a MEX file loader interface to retrieve the decryption key at runtime.

The key file must be in one of the following supported formats:

- Binary 256-bit AES key, with a 32 byte file size
- Hex encoded AES key, with a 64 character file size

The loader MEX file must be an interface with the following arguments:

- `prhs[0]` — Input, char array specified as the static value `'getKey'`
- `prhs[1]` — Input, char array specified as the CTF component UUID
- `plhs[0]` — Output, 32 byte UINT8 numeric array or 64 byte HEX encoded char array

If you do not specify any arguments after `-k`, `mcc` generates a 256-bit AES key and a loader MEX file that can be used for demonstration purposes.

Example: `-k 'file=path\to\encryption.key;loader=path\to\loader_interface.mexw64'`

`-s` — Obfuscate folder structures and file names

Obfuscate folder structures and file names in the deployable archive (`.ctf` file) from the end user. Optionally encrypt additional file types.

The `-s` option directs `mcc` to place user code and data contained in `.m`, `.mlapp`, `.p`, v7.3 `.mat`, MLX, SFX, and MEX files into a user package within the CTF. During runtime, MATLAB code and data is decrypted and loaded directly from the user package rather than extracted to the file system. MEX files are temporarily extracted from the user package before being loaded.

To manually include additional file types in the user package, add each file type in a separate extension tag to the file *`matlabroot`*`/toolbox/compiler/advanced_package_supported_files.xml`.

The following are not supported:

- `ver` function
- Out-of-process MATLAB Runtime (C++ shared library for MATLAB Data Array)
- Out-of-process MEX file execution (`mexhost`, `feval`, `matlab.mex.MexHost`)

MATLAB Runtime

`-R option` — Provide MATLAB Runtime options

`'-logfile,`*`filename`*`'` | `-nodisplay` | `-nojvm` | `'-startmsg,`*`message`*`'` | `'-completemsg,`*`message`*`'` | `-singleCompThread` | `-softwareopengl`

Provide MATLAB Runtime options that are passed to the application at initialization time. This option is only used when building standalone applications or Excel add-ins.

You can specify multiple `-R` options. When you specify multiple `-R` options, they are processed from left to right. For example, specify initialization start and end messages.

```
mcc -R '-startmsg,MATLAB Runtime initialized' -R '-completemsg,Initialization complete'
```

Option	Description	Target
`'-logfile,`*`filename`*`'`	Specify a log file name. The file is created in the application folder at runtime and contains information about MATLAB Runtime initialization and all text piped to the command window. Option must be in single quotes. Use double quotes when executing the command from a Windows Command Prompt.	MATLAB Compiler
`-nodisplay`	Suppress the MATLAB `nodisplay` run-time warning. On Linux, open MATLAB Runtime without display functionality.	MATLAB Compiler
`-nojvm`	Do not use the Java Virtual Machine (JVM).	MATLAB Compiler
`'-startmsg,`*`message`*`'`	Customizable user message displayed at initialization time. For more details, see “Display MATLAB Runtime Initialization Messages” (MATLAB Compiler).	MATLAB Compiler Standalone Applications
`'-completemsg,`*`message`*`'`	Customizable user message displayed when initialization is complete. For more details, see “Display MATLAB Runtime Initialization Messages” (MATLAB Compiler).	MATLAB Compiler Standalone Applications
`-singleCompThread`	Limit MATLAB to a single computational thread.	MATLAB Compiler
`-softwareopengl`	Use Mesa Software OpenGL® for rendering.	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`.

-S — Create single MATLAB Runtime instance

Create a single MATLAB Runtime instance that is shared across all class instances.

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 for the following specific targets.

Target supported by Singleton MATLAB Runtime	Usage Instructions
Excel add-in	Singleton MATLAB Runtime is the default behavior. You do not need to perform other steps.
.NET assembly	Singleton MATLAB Runtime is the default behavior. You do not need to perform other steps.
COM component	• Using the Library Compiler app, click **Settings** and add `-S` to the **Additional parameters passed to MCC** field. • Using `mcc`, pass the `-S` flag.
Java package	

MATLAB Compiler Search Path

`-I folder` — Add folder to search path

folder

Add a new folder to the search path used by MATLAB Compiler during dependency analysis. Each `-I` option appends the folder to the end of the list of paths. For example, the following syntax sets up the search path so that `directory1` is searched first for MATLAB files, followed by `directory2`.

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

This option is important for compilation environments 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.

`-N` — Clear path

Clear the search 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`

`-N` also retains all subfolders in this list that appear on the MATLAB path at compile time. This option 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.

`-p folder` — Conditionally add folders to path

folder

Conditionally 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. You must use this option in conjunction with the option `-N`.

Use the syntax `-N -p directory`, where `directory` is a relative or absolute path to the folder to be included.

- If a folder that is on the original MATLAB path is included with `-p`, 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 that is not on the original MATLAB path is included with `-p`, that folder is ignored. (You can use `-I` to force its inclusion.)

mbuild Options

`-f filename` — `mbuild` options file

file name

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

This option specifically applies to the C/C++ shared libraries, COM, and Excel targets.

`-M options` — `mbuild` compile-time options

Define `mbuild` compile-time options. The *`options`* argument is passed directly to `mbuild`. This option provides a 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	`-M 'COMPFLAGS=$COMPFLAGS /bigobj'`
Windows command prompt	`-M COMPFLAGS="$COMPFLAGS /bigobj"`
Linux and macOS command prompt	`-M CFLAGS='$CFLAGS /bigobj'`

Debugging

`-G` — Include debug symbols

Include debugging symbol information for the C/C++ code generated by MATLAB Compiler SDK. This option 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.

`-K` — Keep partial output

Keep partial 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.

`-v` — Display verbose output

Display verbose output. Output displays the compilation steps, including:

- MATLAB Compiler version number
- Source file names as they are processed
- Names of the generated output files as they are created
- Invocation of `mbuild`

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

-w option[:warning] — Control warning messages
list | enable | disable | error | on | off

Control the display of warning messages.

Syntax	Description
`-w list`	List the compile-time warnings that have abbreviated identifiers along with their status.
`-w enable[:<warning>]`	Enable specific compile-time warnings associated with `<warning>`. Omit the optional `<warning>` to apply the `enable` action to all compile-time warnings.
`-w disable[:<warning>]`	Disable specific compile-time warnings associated with `<warning>`. Omit the optional `<warning>` to apply the `disable` action to all compile-time warnings.
`-w error[:<warning>]`	Treat specific compile-time and runtime warnings associated with `<warning>` as an error. Omit the optional `<warning>` to apply the `error` action to all compile-time and runtime warnings.
`-w on[:<warning>]`	Turn on runtime warnings associated with `<warning>`. Omit the optional `<warning>` to apply the `on` action to all runtime warnings. This option is enabled by default.
`-w off[:<warning>]`	Turn off runtime warnings for specific error messages defined by `<warning>`. Omit the optional `<warning>` to apply the `off` action to all runtime warnings.

The `<warning>` argument can be either a full identifier, such as `Compiler:compiler:COM_WARN_OPTION_NOJVM`, or one of the abbreviated identifiers listed by `-w list`.

You can display the full identifier that corresponds to a warning by issuing the following statement in your MATLAB code after the warning takes place.

```
[msg, warnID] = lastwarn
```

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

For example, disable all warnings except `repeated_file`.

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

You can also turn warnings on or off globally. For example, to turn off warnings for all deployed applications, specify the following in `startup.m` using `isdeployed`.

```
if isdeployed
    warning off
end
```

Limitations

- `mcc` cannot create web apps. To create web apps, use the **Web App Compiler** app or the `compiler.build.webAppArchive` function.

Tips

- On Windows, you can generate a system-level file version number for your target file by appending `version=`*`version_number`* to the target generating `mcc` syntax. For an example, see “Create Excel Add-In (Windows only)” (MATLAB Compiler).

  `version_number` — Specifies the version of the target file as *`major.minor.bug.build`* in the file system. You are not required to specify a version number. If you do not specify a version number, `mcc` sets the version number, by default, to `1.0.0.0`.

  - *`major`* — Specifies the major version number. If you do not specify a version number, `mcc` sets *`major`* to `1`.
  - *`minor`* — Specifies the minor version number. If you do not specify a version number, `mcc` sets *`minor`* to `0`.
  - *`bug`* — Specifies the bug fix maintenance release number. If you do not specify a version number, `mcc` sets *`bug`* to `0`.
  - *`build`* — Specifies build number. If you do not specify a version number, `mcc` sets *`build`* to `0`.

  This functionality is supported for standalone applications and Excel add-ins in MATLAB Compiler.

  This functionality is supported for C shared libraries, C++ shared libraries, COM components, .NET assemblies, and Excel add-ins for MATLAB Production Server in MATLAB Compiler SDK.

Version History

Introduced before R2006a

See Also

`ismcc` | `isdeployed` | `deploytool` | `compiler.build.Results`

Topics

“mcc Command Arguments Listed Alphabetically” (MATLAB Compiler)
“How Does MATLAB Deploy Functions?” (MATLAB Compiler)
“Write Deployable MATLAB Code” (MATLAB Compiler)
“Access Files in Packaged Applications” (MATLAB Compiler)

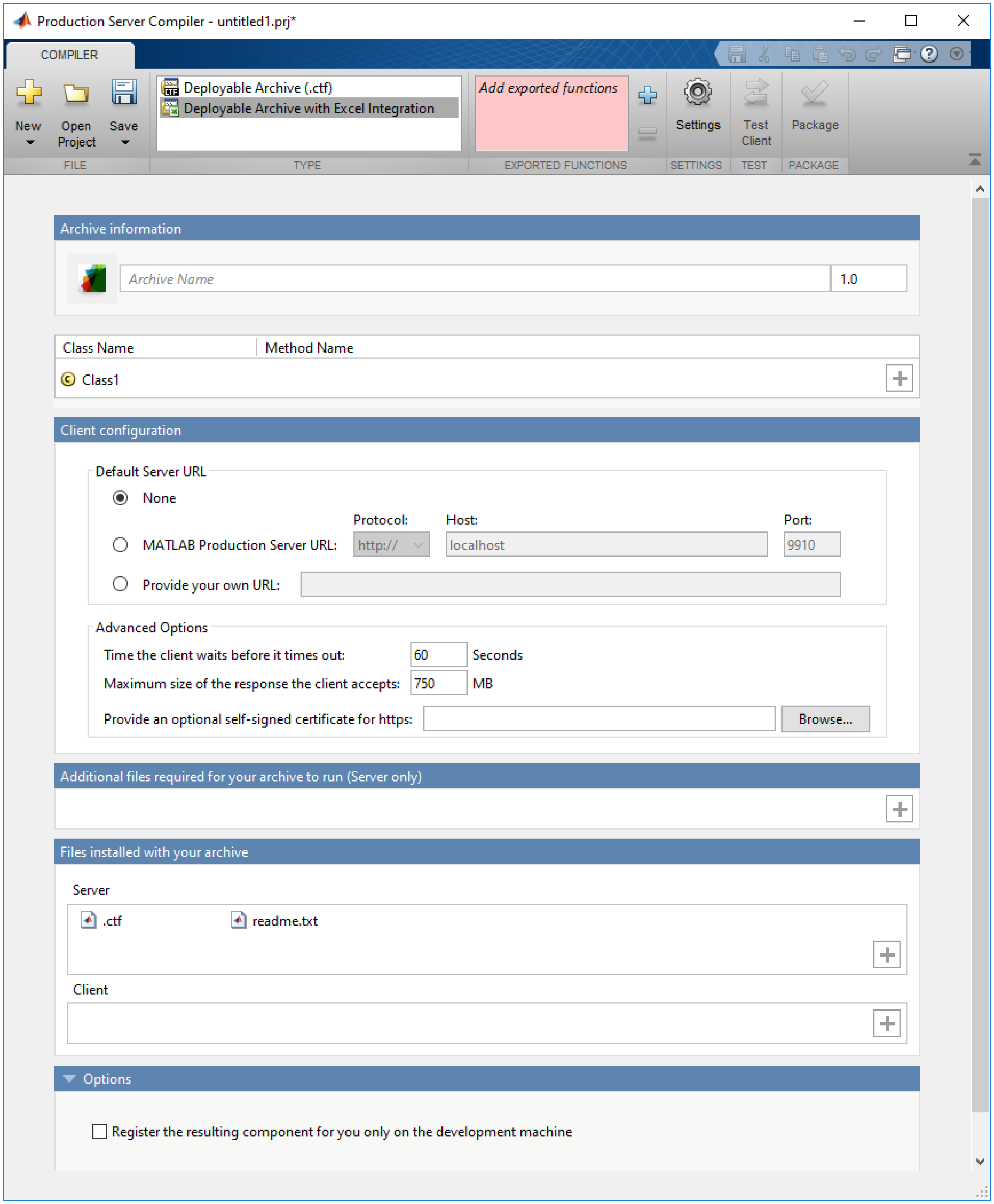
Apps

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.



Deployable Archive (.ctf)
Deployable Archive with Excel Integration

Add exported functions

Archive information

Archive Name 1.0

Class Name	Method Name
Class1	

Client configuration

Default Server URL

(● None)

○ MATLAB Production Server URL: Protocol: http:// Host: localhost Port: 9910

○ Provide your own URL:

Advanced Options

Time the client waits before it times out: 60 Seconds

Maximum size of the response the client accepts: 750 MB

Provide an optional self-signed certificate for https: Browse...

Additional files required for your archive to run (Server only)

Files installed with your archive

Server

.ctf readme.txt

Client

Options

☐ Register the resulting component for you only on the development machine

Open the Production Server Compiler App

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

Examples

- “Create Deployable Archive for MATLAB Production Server” on page 1-2
- “Create and Install a Deployable Archive with Excel Integration for MATLAB Production Server”
- “Test Client Data Integration Against MATLAB” (MATLAB Compiler SDK)

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

Enter `productionServerCompiler`.

Alternatively, enter `deploytool` and click **Production Server Compiler**.

Version History

Introduced in R2013b

See Also

`deploytool` | `compiler.build.productionServerArchive` | `mcc`

Topics

"Create Deployable Archive for MATLAB Production Server" on page 1-2
"Create and Install a Deployable Archive with Excel Integration for MATLAB Production Server"
"Test Client Data Integration Against MATLAB" (MATLAB Compiler SDK)

6

Persistence

Data Caching Basics

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.

Typical Workflow for Data Caching

Steps	Command Line	Dashboard
1. Create file `mps_cache_config`	Manually create a JSON file and place it in the `config` folder of the server instance. Do not include the `.json` extension in the filename.	Automatically created.
2. Start persistence service	Use `mps-cache` to start a persistence service. For testing purposes, you can create a persistence service controller object using `mps.cache.control`.	• Create a persistence service. • Add the persistence service to a server instance using a connection name. • Start the persistence service. • Attach the connection associated with a persistence service to a server instance.
3. Create a data cache	Use `mps.cache.connect` to create a data cache.	Use `mps.cache.connect` to create a data cache.

Configure Server to Use Redis

Create Redis Configuration File

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` (no `.json` extension) and place it in the `config` folder of the server instance. If you use the dashboard to manage an on-premises server instance and for server deployments on the cloud, the `mps_cache_config` file is automatically created on server creation.

`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.

Install WSL for Server Instances Running on Windows

If your MATLAB Production Server instance runs on a Windows machine, you require additional configuration. The following configuration is not required for server instances that run on Linux and macOS.

- You need to install Windows Subsystem for Linux (WSL). For details on installing WSL, see Microsoft documentation.
- If the MATLAB Production Server software is installed on a network drive, you must mount the network drive in WSL.

Example: Increment Counter Using 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 in MATLAB Compiler SDK

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 Using MATLAB Production Server

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 for MATLAB Function Execution" from the MATLAB desktop.

```
rhs = {['myCache'],['myRedisConnection']};
body = mps.json.encoderequest(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

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

More About

- "Manage Application State in Deployed Archives"

Manage Application State in Deployed Archives

This example shows how to manage persistent data in application archives deployed to MATLAB Production Server. It uses the MATLAB Production Server "RESTful API for MATLAB Function Execution" 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 for MATLAB Production Server".

1. "Step 1: Write MATLAB Code that uses Persistence Functions" on page 6-5
2. "Step 2: Run Example in Testing Workflow" on page 6-9
3. "Step 3: Run Example in Deployment Workflow" on page 6-10

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,'ContentType','raw');
    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, listDestinations_OPTIONS);
    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','raw');
    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), ...
            mps.json.encoderequest(start_destinations_DATA,'nargout',nargout), findRoute_OPTIONS);
    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

    addDestination_OPTIONS = weboptions('MediaType','application/json','Timeout',60,'ContentType','raw');
    addDestination_HOST = app.HostEditField.Value;
    addDestination_PORT = app.PortEditField.Value;
    name_destinations_distances_DATA = {};
    if nargin > 0, name_destinations_distances_DATA = [ name_destinations_distances_DATA { name } ]; end
    if nargin > 1, name_destinations_distances_DATA = [ name_destinations_distances_DATA { destinations } ]; end
    if nargin > 2, name_destinations_distances_DATA = [ name_destinations_distances_DATA { distances } ]; end
    count_JSON = webwrite(sprintf('http://%s:%d/TravelingSalesman/addDestination', ...
        addDestination_HOST,addDestination_PORT), ...
        mps.json.encoderequest(name_destinations_distances_DATA,'nargout',nargout), addDestination_OPTIONS);
    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
```

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`. For details, see "Configure Server to Use Redis" on page 6-2. 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`.

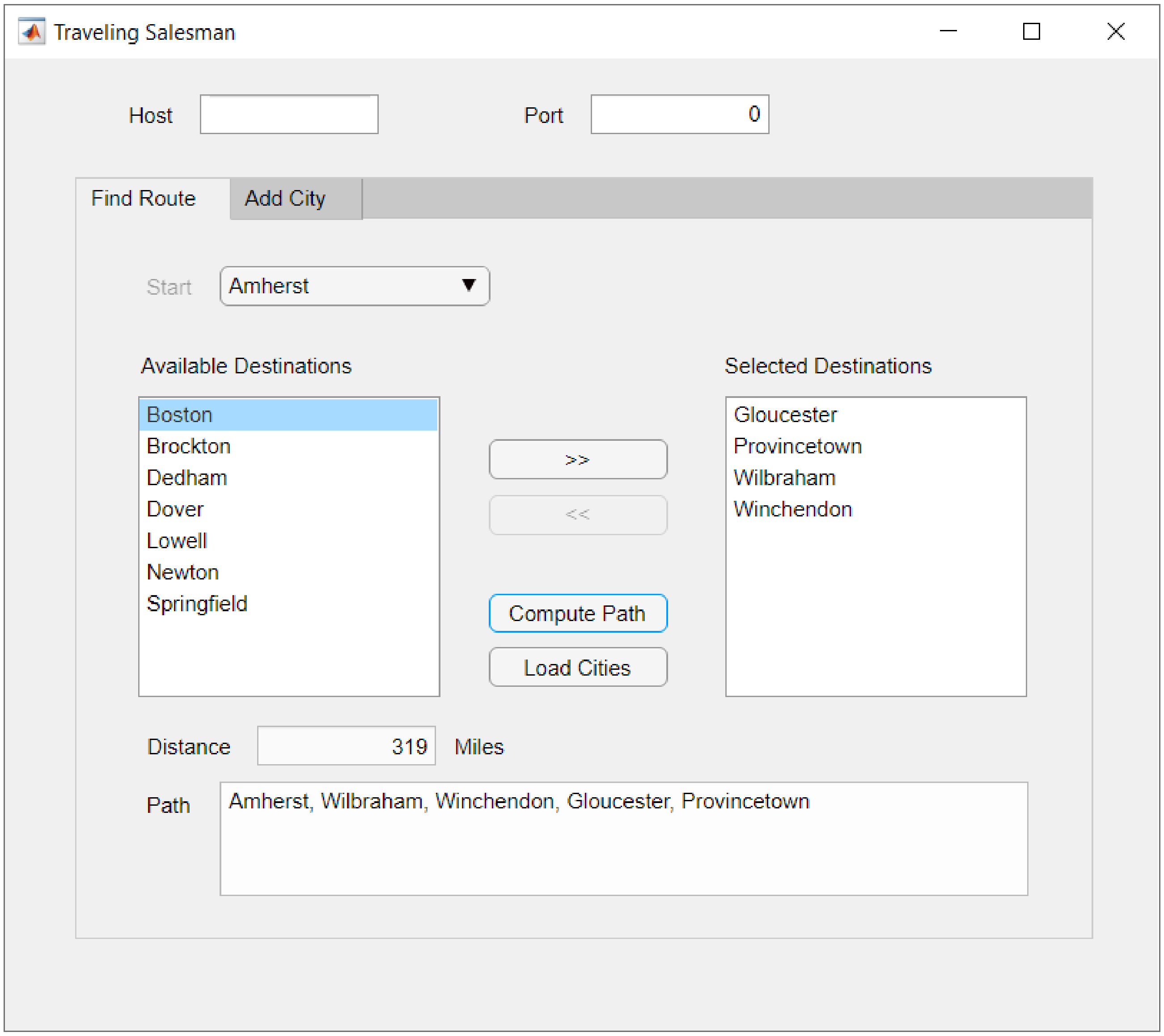
```
>> start(ctrl);
```

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.



Amherst, Wilbraham, Winchendon, Gloucester, Provincetown

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 the 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 Using Command Line”. 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.

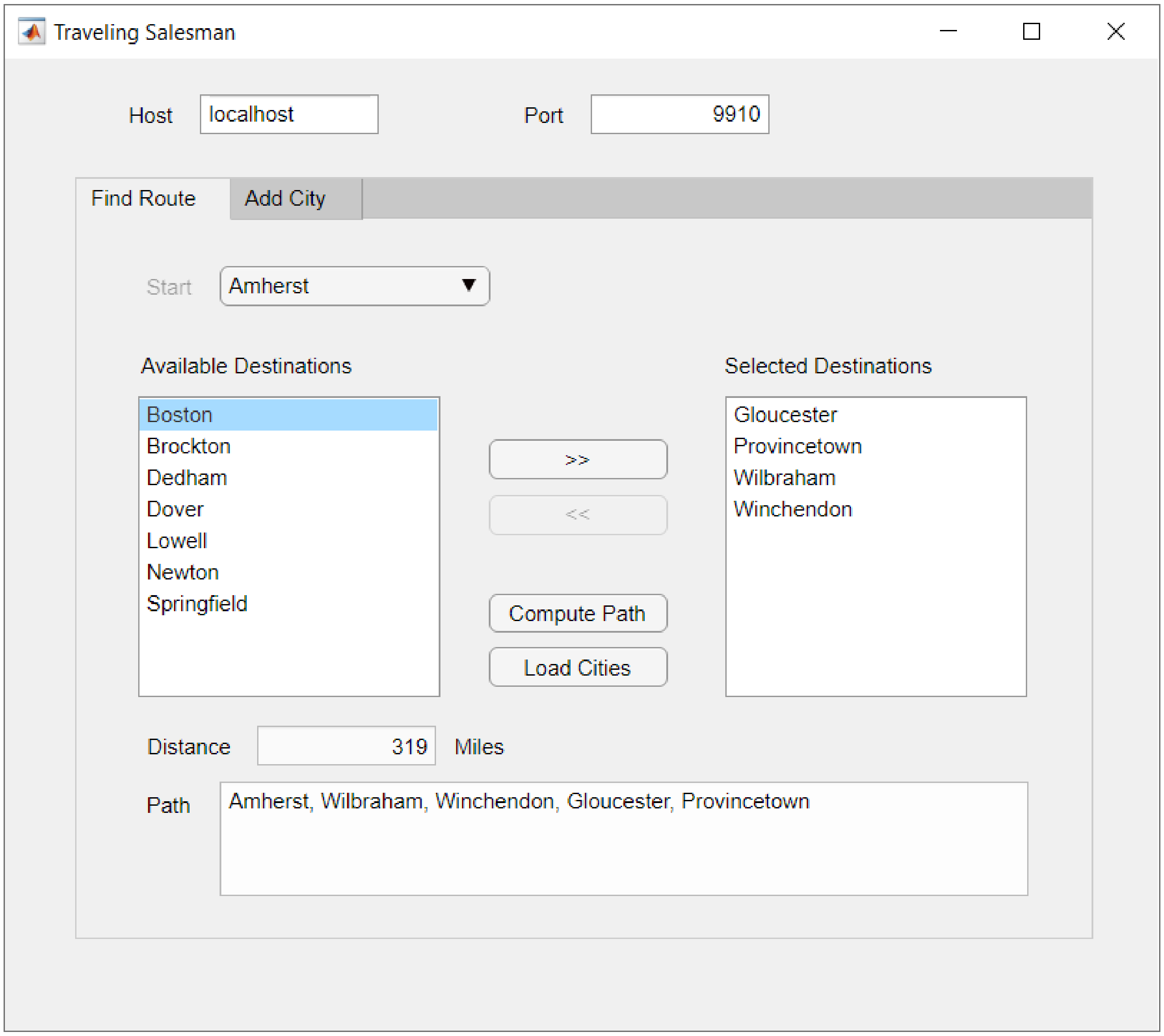
6. Test the app

   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.



Host: localhost
Port: 9910
Start: Amherst

Available Destinations: Boston, Brockton, Dedham, Dover, Lowell, Newton, Springfield

Selected Destinations: Gloucester, Provincetown, Wilbraham, Winchendon

Distance: 319 Miles

Path: Amherst, Wilbraham, Winchendon, Gloucester, Provincetown

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

See Also

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

More About

- “Data Caching Basics”

Handle Custom Routes and Payloads in HTTP Requests

Web request handlers for MATLAB Production Server provide flexible client-server communication.

- Client programmers can send custom HTTP headers and payloads in RESTful requests to the server.
- Server administrators can provide flexible mapping of the request URLs to deployed MATLAB functions.
- Server administrators can provide static file serving.
- MATLAB programmers can return custom HTTP headers, HTTP status codes, HTTP status messages, and payloads in functions deployed to MATLAB Production Server.

To use web request handlers, you write the MATLAB function that you deploy to the server in a specific way and specify custom URL routes in a JSON file on the server.

Write MATLAB Function for Web Request Handler

To work as a web request handler, the MATLAB function that you deploy to the server must accept one input argument that is a scalar structure array, and return one output argument that is a scalar structure array.

The structure in the function input argument provides information about the client request. Clients can send custom HTTP headers and custom payloads. There are no data format restrictions on the payload that the deployed function can accept. For example, the function can accept raw data in binary or ASCII formats, CSV data, or JSON data that is not in the schema specified by the MATLAB Production Server RESTful API. Clients can also use the `Transfer-Encoding: chunked` header to send data in chunks. In chunked transfer encoding, though the server receives payload in chunks, the input structure receives payload data in entirety.

The structure in the function input argument contains the following fields:

Field Name	Data Type	Dimensions	Description
`ApiVersion`	`double`	1 x 3	Version of the input structure schema in the format *`<major>`* *`<minor>`* *`<fix>`*
`Body`	`uint8`	1 x N	Request payload
`Headers`	`cell`	N x 2	HTTP request headers Each element in the cell array represents a header. Each element is a key-value pair, where the key is of type `char` and the value can be of type `char` or `double`.
`HttpVersion`	`double`	1 x 2	HTTP version in the format *`<major>`* *`<minor>`*

Field Name	Data Type	Dimensions	Description
`Method`	`char`	1 x N	HTTP request method
`Path`	`char`	1 x N	Path of request URL

Since the deployed MATLAB function can accept custom headers and payloads in RESTful requests, you can vary the behavior of the MATLAB function depending on the request header data. You can use the structure in the function output argument to return a response with custom HTTP headers and payload. Server processing errors, if any, override any custom HTTP headers that you might set. If a MATLAB error occurs, the server returns an HTTP `500 Internal Server Error` response. All fields in the structure are optional.

The structure in the output argument can contain the following fields:

Field Name	Data Type	Dimensions	Description
`ApiVersion`	`double`	1 x 3	Version of the output structure schema in the format *`<major> <minor> <fix>`*
`Body`	`uint8`	1 x N	Response payload
`Headers`	`cell`	N x 2	HTTP response headers Each element in the cell array represents a header. Each element is a key-value pair, where the key is of type `char` and the value can be of type `char` or `double`.
`HttpCode`	`double`	1 x 1	HTTP status code
`HttpMessage`	`char`	1 x N	HTTP status message

Configure Server for URL Routes

Custom URL routes allow the server to map the path in request URLs to any deployable archive and MATLAB function deployed on the server.

To specify the mapping of a request URL to a deployed MATLAB function, you use a JSON file present on the server. The default name of the file is `routes.json` and its default location is in the *`$MPS_INSTALL`*`/config` directory. You can change the file name and its location by changing the value of the `--routes-file` property in the `main_config` server configuration file. You must restart the server after making any updates to `routes.json`.

When the server starts, it tries to read the `routes.json` file. If the file does not exist or contains errors, the server does not start, and writes an error message to the `main.log` file present in the directory that the log-root property specifies.

The default `routes.json` contains a `version` field with a value of `1.0.0`, and an empty `pathmap` field. `version` specifies the schema version of the file. You do not need to change its value. To allow custom routes, edit the file to specify mapping rules in the `pathmap` array. In the `pathmap` array, you can specify multiple objects, where each object corresponds to a URL route.

Following is the schema of `routes.json`.

```
{
  "version": "1.0.0",
   "pathmap": [
       {
           "match": "<regular_expression>",
           "webhandler": {
               "component": "<name_of_deployable_archive>",
               "function": "<name_of_deployed_function>"
           }
       },
    {
           "match": "<regular_expression>",
           "webhandler": {
               "component": "<name_of_deployable_archive>",
               "function": "<name_of_deployed_function>"
           }
       }
    ]
}
```

To specify a URL mapping rule, use the `match` and `webhandler` fields from the `pathmap` array.

- In the `match` field, specify a regular expression that uses ECMAScript grammar to match the path in a request URL.
  - If the request URL matches multiple regular expressions in the `match` field, the first match starting from the beginning of the file is selected.
  - The regular expression patterns are considered a match if any substring of the request URL is a match. For example, the pattern `a/b` matches `a/b`, `/a/b`, and `/x/a/b/y`. However, you can use the regular expression anchors, `^` and `$`, to match positions before or after specific characters. For example, the pattern `^a/b$` only matches `a/b`.
  - You can specify regular expressions that match query parameters in the request URL. However, asynchronous request execution using the MATLAB Production Server RESTful API is not supported. Request execution is synchronous. For more information about the MATLAB Production Server RESTful API, see "RESTful API for MATLAB Function Execution".
- In the `webhandler` field, use the `component` field to specify the name of the deployable archive and the `function` field to specify the name of the deployed function for the request URL to execute.

End-to-End Setup for Web Request Handler

This example assumes you have a server instance running at the default host and port, localhost:9910. For information on starting a server, see "Start Server Instance Using Command Line".

1 Write a MATLAB function for the web request handler.

   The following code shows a MATLAB function that uses an input argument structure `request`, whose fields provide information about the request headers and body. The function also constructs and returns a structure `response`, whose fields contain a success HTTP code and status message, custom headers, and a message body.

```
function response = hellowh(request)
    disp(request);
    disp('request.Headers:');
    disp(request.Headers);
    bodyText = char(request.Body);
    disp('request.Body:');
    if length(bodyText) > 100
        disp(bodyText(1:100));
        disp('...');
    else
        disp(bodyText);
    end

    response = struct('ApiVersion', [1 0 0], ...
                      'HttpCode', 200, ...
                      'HttpMessage', 'OK', ...
                      'Headers', {{ ...
                        'Server' 'WebFunctionTest/1'; ...
                        'X-MyHeader' 'foobar'; ...
                        'X-Request-Body-Len' sprintf('%d', length(request.Body)); ...
                        'Content-Type' 'text/plain'; ...
                      }},...
                      'Body', uint8('hello, world'));

    disp(response);
    disp('response.Headers:');
    disp(response.Headers);
end
```

2 Package the function into a deployable archive.

   The following command compiles the `hellowh.m` function into a deployable archive, `whdemo.ctf`. For other ways to create deployable archives, see “Create Deployable Archive for MATLAB Production Server” on page 1-2.

   ```
   mcc -v -U -W 'CTF:whdemo' hellowh.m
   ```

3 Deploy the archive, `whdemo`, to the server. For more information, see “Deploy Archive to MATLAB Production Server”.

4 Edit the `routes.json` file on the server to map a client request to the deployed function. Restart the server instance for the changes to take effect. See mps-restart.

   The following file maps any client request that contains `MyDemo` in the request URL to the `hellowh` function in the `whdemo` archive deployed to the server.

```
{
  "version": "1.0.0",
    "pathmap": [
        {
            "match": "^/MyDemo/.*",
            "webhandler": {
                "component": "whdemo",
                "function": "hellowh"
            }
        }
    ]
}
```

5 Use a client of your choice to invoke the deployed function.

   The following command uses cURL to invoke the deployed function from the system command line.

   ```
   curl -v http://localhost:9910/MyDemo/this/could/be/any/path?param=YES
   ```

You see the following output at the system command line:

```
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 9910 (#0)
> GET /MyDemo/this/could/be/any/path?param=YES HTTP/1.1
> Host: localhost:9910
> User-Agent: curl/7.55.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: WebFunctionTest/1
< X-MyHeader: foobar
< X-Request-Body-Len: 0
< Content-Type: text/plain
< Content-Length: 12
< Connection: Keep-Alive
<
hello, world* Connection #0 to host localhost left intact
```

See Also
files-root

Related Examples
- “Test Web Request Handlers” (MATLAB Compiler SDK)
- “Create Deployable Archive for MATLAB Production Server” on page 1-2
- “Deploy Archive to MATLAB Production Server”

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

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

Version History
Introduced in R2018b

See Also
`mps.cache.Controller`

Topics
"Data Caching Basics" on page 6-2

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 to a 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 MATLAB session to persistence service that is already running
detach	Disconnect MATLAB session from 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'
```

Version History

Introduced in R2018b

See Also

`mps.cache.DataCache`

Topics

"Data Caching Basics" on page 6-2

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`.

Version History
Introduced in R2018b

See Also
`mps.cache.DataCache`

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.

Version History

Introduced in R2018b

See Also

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

Topics

"Data Caching Basics" on page 6-2

attach

Package: `mps.cache`

Connect MATLAB session to 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 the system command line using `mps-cache` or using the dashboard. Assuming your 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)`

Version History

Introduced in R2018b

See Also

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

Topics

"Data Caching Basics" on page 6-2

detach

Package: `mps.cache`

Disconnect MATLAB session from 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)`

Version History

Introduced in R2018b

See Also

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

Topics
"Data Caching Basics" on page 6-2

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)`

Version History
Introduced in R2018b

See Also
`stop` | `restart` | `attach` | `detach`

Topics
"Data Caching Basics" on page 6-2

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)`

Version History

Introduced in R2018b

See Also

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

Topics
"Data Caching Basics" on page 6-2

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)`

Version History

Introduced in R2018b

See Also

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

Topics
"Data Caching Basics" on page 6-2

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 yourMATLAB 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)`

Version History

Introduced in R2018b

See Also

`start` | `stop` | `restart`

Topics

"Data Caching Basics" on page 6-2

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 yourMATLAB 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)`

Version History

Introduced in R2018b

See Also

`start` | `stop` | `restart`

Topics
"Data Caching Basics" on page 6-2

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)`.

Version History

Introduced in R2018b

See Also

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

Topics

"Data Caching Basics" on page 6-2

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 =

  5×2 table

       Keys            Values
    ___________     ____________

    'keyFive'       [5×5 double]
    'keyFour'       [1×2 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

Version History

Introduced in R2018b

See Also

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

Topics
"Data Caching Basics" on page 6-2

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.

Version History

Introduced in R2018b

See Also

`retain` | `purge` | `clear` | `keys` | `remove`

Topics

"Data Caching Basics" on page 6-2

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.

Version History

Introduced in R2018b

See Also

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

Topics
"Data Caching Basics" on page 6-2

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.

Version History

Introduced in R2018b

See Also

`get` | `keys` | `put`

Topics
“Data Caching Basics” on page 6-2

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.

Version History

Introduced in R2018b

See Also

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

Topics
"Data Caching Basics" on page 6-2

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.

Version History

Introduced in R2018b

See Also

`isKey` | `bytes` | `get` | `length` | `put`

Topics
“Data Caching Basics” on page 6-2

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.

Version History

Introduced in R2018b

See Also

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

Topics
"Data Caching Basics" on page 6-2

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 —

Version History

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.

Version History

Introduced in R2018b

See Also

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

Topics
"Data Caching Basics" on page 6-2

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 by `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 =

  5×2 table

       Keys            Values
    __________      ____________

    'keyFive'       [5×5 double]
    'keyFour'       [1×2 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)}`

Version History

Introduced in R2018b

See Also

`keys` | `get` | `bytes` | `length` | `remove` | `clear`

Topics

“Data Caching Basics” on page 6-2

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 =

  3×2 table

      Keys          Values    
    _________    ____________

    'keyFive'    [5×5 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.

Version History

Introduced in R2018b

See Also

`put` | `keys` | `get` | `purge` | `retain` | `clear`

Topics
"Data Caching Basics" on page 6-2

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.

Version History

Introduced in R2018b

See Also

`flush` | `purge` | `remove` | `clear`

Topics
“Data Caching Basics” on page 6-2

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2018b

See Also

`acquire` | `own` | `release` | `mps.sync.TimedRedisMutex` | `mps.sync.TimedMATFileMutex`

Topics

“Data Caching Basics” on page 6-2

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

mps.sync.mutex	Create a persistence service mutex
acquire	Acquire advisory lock on persistence service mutex
own	Check ownership of advisory lock on a persistence service mutex object
release	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'
```

Version History

Introduced in R2018b

See Also

`mps.sync.mutex` | `mps.sync.TimedMATFileMutex` | `acquire` | `own` | `release`

Topics

"Data Caching Basics" on page 6-2

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

mps.sync.mutex	Create a persistence service mutex
acquire	Acquire advisory lock on persistence service mutex
own	Check ownership of advisory lock on a persistence service mutex object
release	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'
```

Version History

Introduced in R2018b

See Also

mps.sync.mutex | mps.sync.TimedRedisMutex | acquire | own | own | release | release

Topics

"Data Caching Basics" on page 6-2

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.mutex('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.

Version History

Introduced in R2018b

See Also

`mps.sync.mutex` | `own` | `release` | `mps.sync.TimedRedisMutex` | `mps.sync.TimedMATFileMutex`

Topics
"Data Caching Basics" on page 6-2

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.mutex('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.

Version History

Introduced in R2018b

See Also

`mps.sync.mutex` | `acquire` | `release` | `mps.sync.TimedRedisMutex` | `mps.sync.TimedMATFileMutex`

Topics

"Data Caching Basics" on page 6-2

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.mutex('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.

Version History

Introduced in R2018b

See Also

`mps.sync.mutex` | `acquire` | `own` | `mps.sync.TimedRedisMutex` | `mps.sync.TimedMATFileMutex`

Topics
“Data Caching Basics” on page 6-2

MATLAB Client

- “Connect MATLAB Session to MATLAB Production Server” on page 8-2
- “Execute Deployed MATLAB Functions” on page 8-5
- “Configure Client-Server Communication” on page 8-11
- “Application Access Control” on page 8-14
- “Execute Deployed Functions Using HTTPS” on page 8-17
- “Manage Add-Ons” on page 8-20
- “Deploy Add-Ons” on page 8-25

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 8-5.

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 8-5

- “Get and Manage Add-Ons” (MATLAB)
- “Execute Deployed Functions Using HTTPS” on page 8-17

Execute Deployed MATLAB Functions

In this section...

"Install MATLAB Client for MATLAB Production Server" on page 8-5

"Deploy MATLAB Function on Server" on page 8-5

"Install MATLAB Production Server Add-On for the Deployable Archive" on page 8-6

"Manage Installed Add-On" on page 8-8

"Invoke Deployed MATLAB Function" on page 8-9

This example shows how to use MATLAB Client for MATLAB Production Server to invoke a MATLAB function deployed to 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**.

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 1-2 and “Deploy Archive to MATLAB Production Server”.

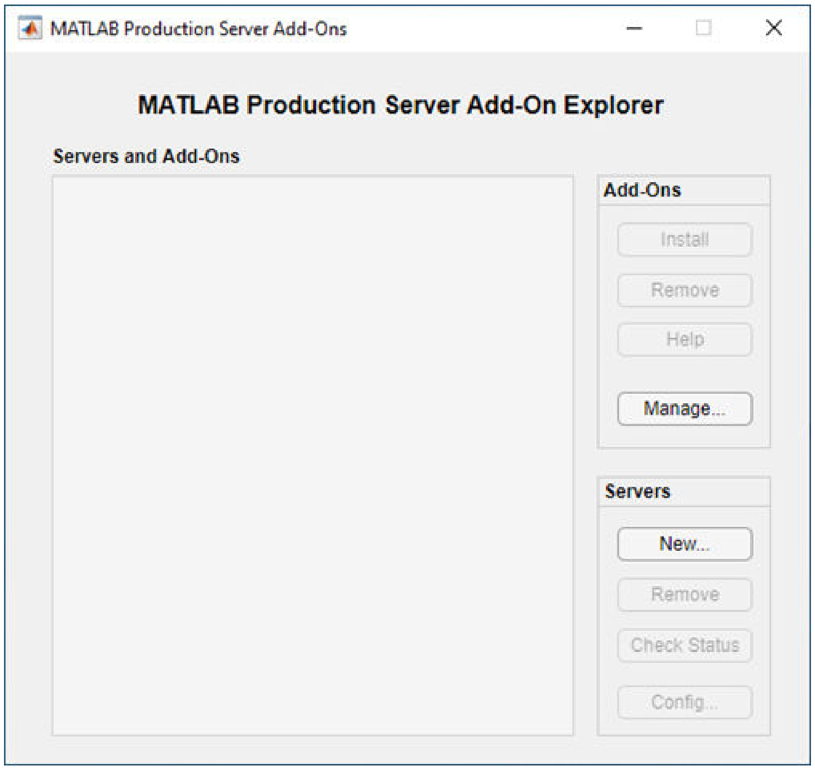
Install MATLAB Production Server Add-On for the Deployable Archive

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
```

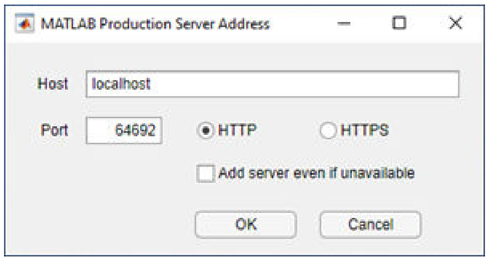


Add Server Information

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.


MATLAB Production Server Address

Host: localhost

Port: 64692 [✓ HTTP] HTTPS

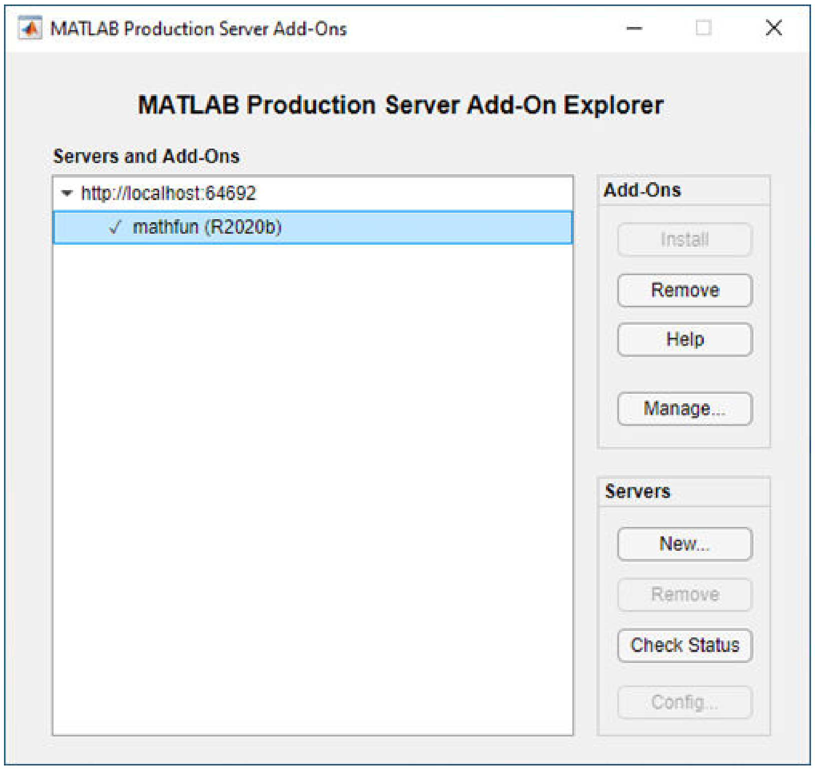
☐ Add server even if unavailable

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.

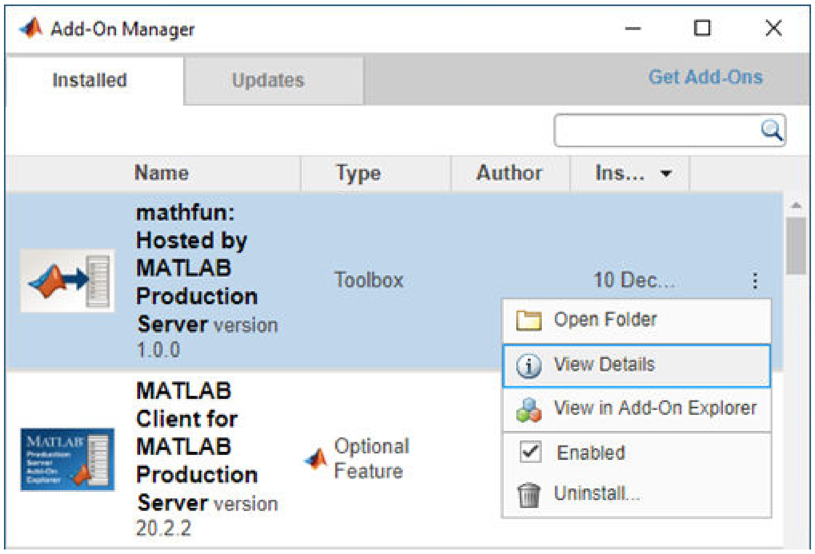
Install the `mathfun` add-on to make the MATLAB function `mymagic` from the deployable archive `mathfun` available in your MATLAB client programs.

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.



Name	Type	Author	Ins...
mathfun: Hosted by MATLAB Production Server version 1.0.0	Toolbox		10 Dec...
MATLAB Client for MATLAB Production Server version 20.2.2	Optional Feature		

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.

You can install multiple add-ons that have the same name but are hosted on different servers. The proxy functions that the add-ons create appear on the MATLAB search path. When you call a proxy function, the function with the same name that appears nearest to the top of the MATLAB search path is invoked. For more information about the MATLAB search path, see “What Is the MATLAB Search Path?” (MATLAB).

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 8-11.

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 8-2
- “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 `PRODSERVER_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": "e3325lo6-4297-47d2-9ec8-9df64195fce3",
          "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 `locahost: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 8-14.

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 8-5
- “Execute Deployed Functions Using HTTPS” on page 8-17
- “Application Access Control” on page 8-14
- “Manage Add-Ons” on page 8-20

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 8-5.

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

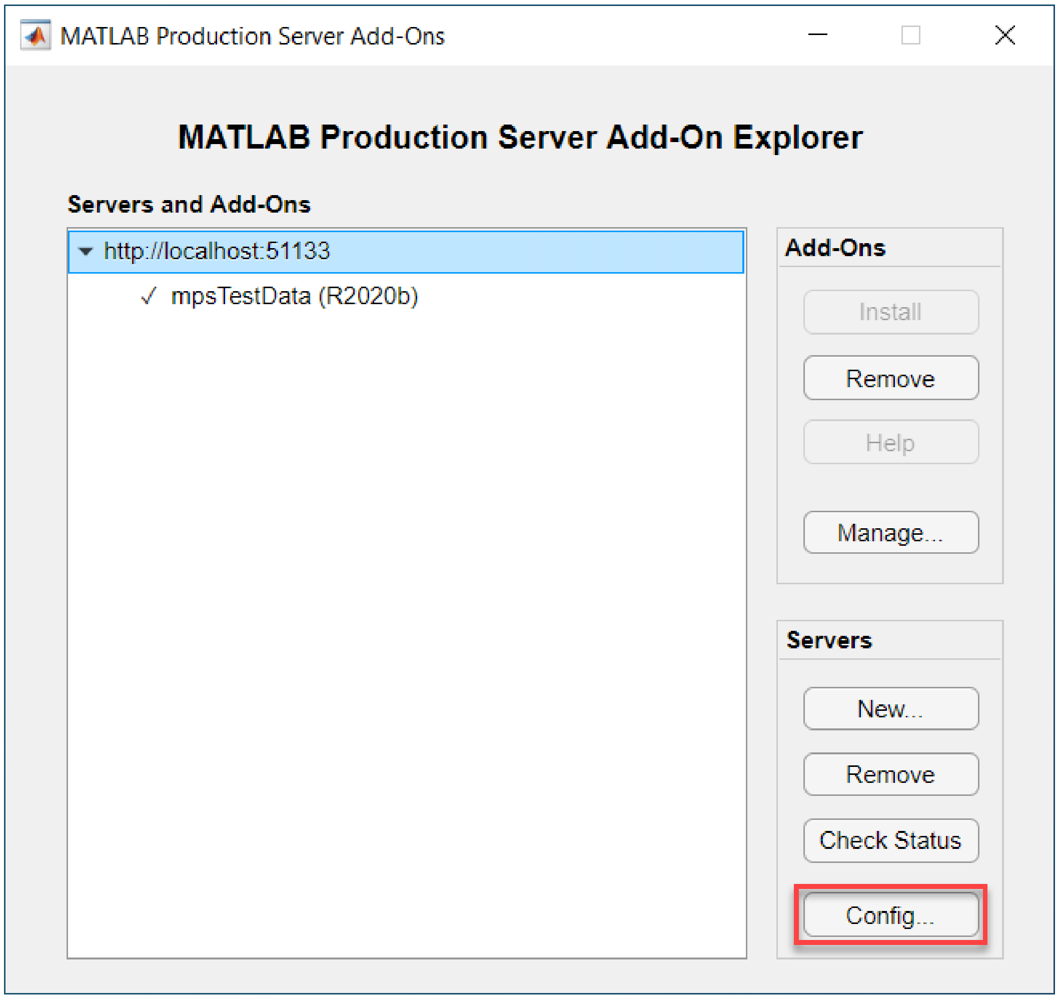
If you want to specify your own bearer token, you can use the `prodserver.addon.accessTokenPolicy` function to do so.

```
>> prodserver.addon.accessTokenPolicy('localhost',51133,'your_access_token')
```

Set Access Token Policy Using MATLAB Production Server Add-On Explorer App

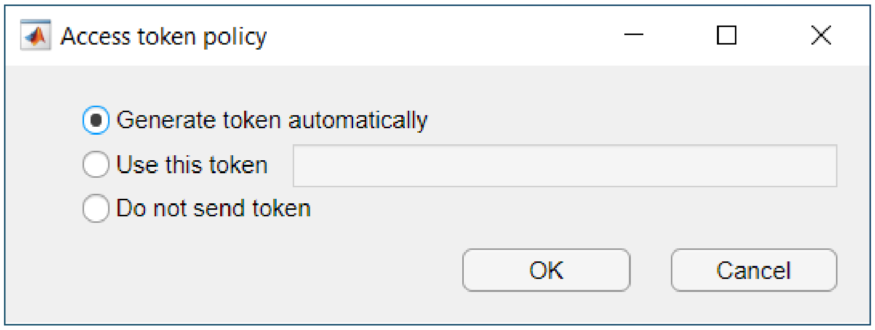
You can use the **MATLAB Production Server Add-On Explorer** to switch between using a system-generated bearer token or specifying your own bear token.

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.



Access token policy

[✓ Generate token automatically]

Use this token

Do not send token

See Also

`prodserver.addon.accessTokenPolicy`

More About

- “Application Access Control”
- “Execute Deployed MATLAB Functions” on page 8-5

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 **Connection is Secure** for a CA-signed SSL certificate or **Certificate is not Valid** for a self-signed SSL certificate. Then click **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. For details on adding the certificate using MMC, see Add Certificates to the Certificate Store on the Microsoft website.

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_name.certificate_extension')
```

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.

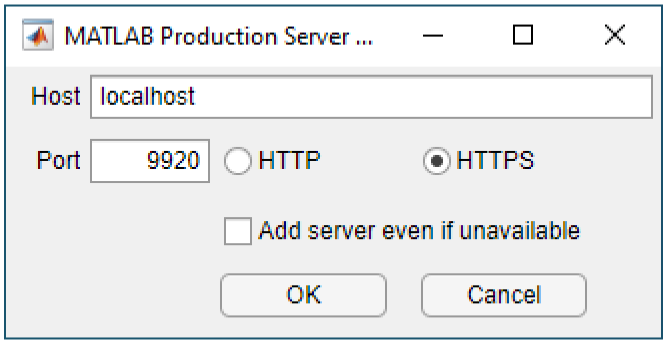
Install Add-On Using Command Line

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

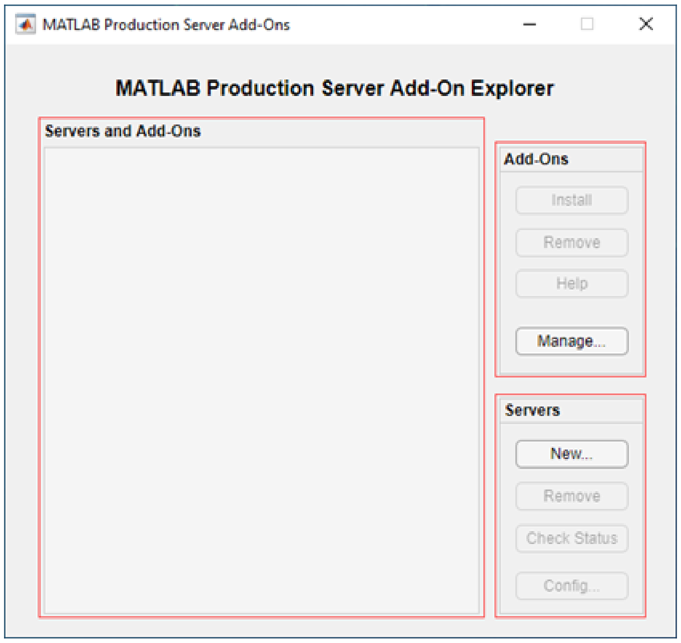
`prodserver.addon.install` | `prodserver.addon.get` | `prodserver.addon.set` | `prodserver.addon.accessTokenPolicy`

External Websites

- “Execute Deployed MATLAB Functions” on page 8-5
- “Application Access Control” on page 8-14

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 `prodserver.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.

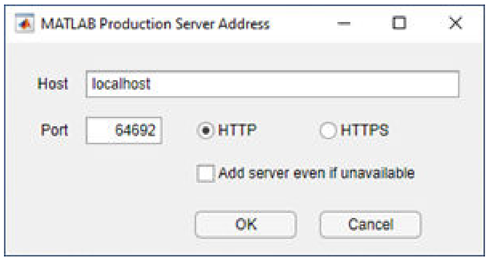
Install Add-Ons

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.


MATLAB Production Server Address

Host: localhost

Port: 64692 (● HTTP ○ HTTPS)

☐ Add server even if unavailable

To check the server status, select the server from the **Servers and Add-Ons** section, then click **Check Status**.

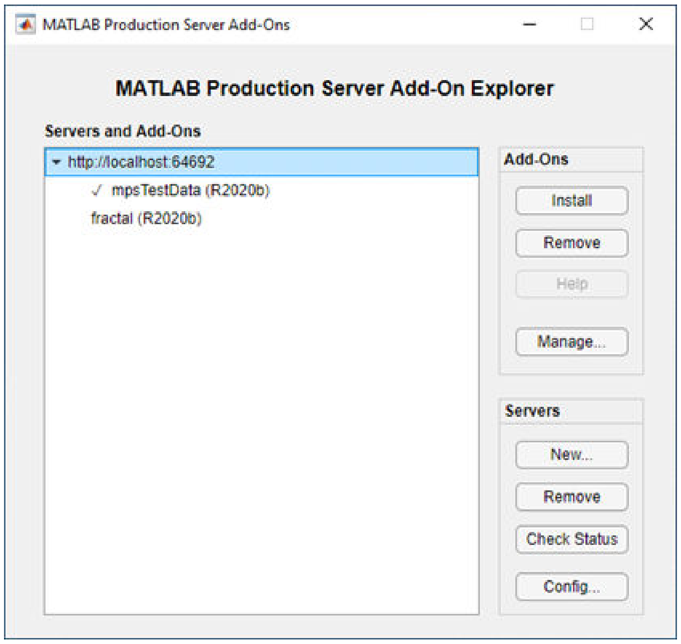
To remove a server, select the server from the **Servers and Add-Ons** section, then click **Remove**. Removing a server also removes the add-ons installed from the server.

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 sections 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.


MATLAB Production Server Add-Ons

MATLAB Production Server Add-On Explorer

Servers and Add-Ons

http://localhost:64692
✓ mpsTestData (R2020b)
fractal (R2020b)

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 8-5 and "Execute Deployed Functions Using HTTPS" on page 8-17.

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.

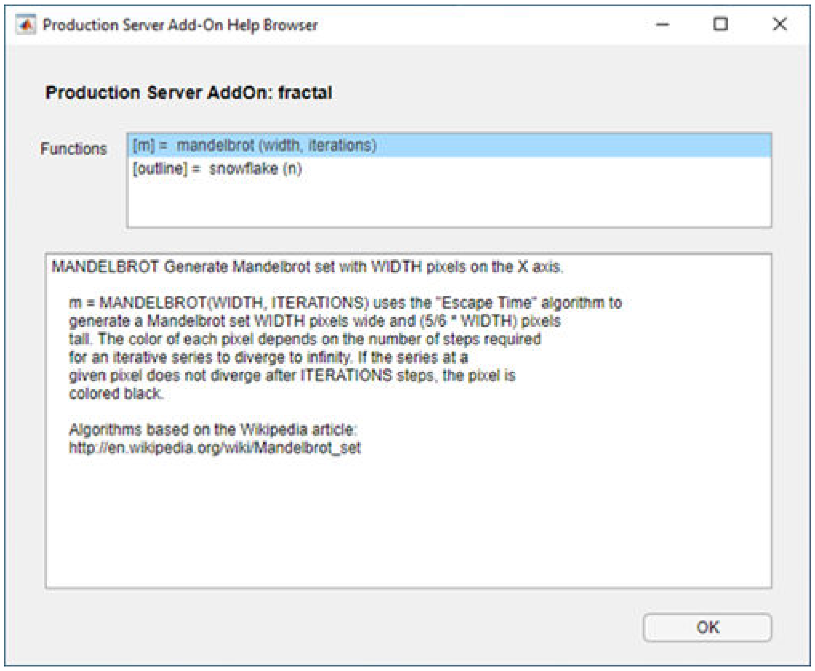
For information about uninstalling add-ons from the MATLAB command prompt, see `prodserver.addon.uninstall`.

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.



Production Server AddOn: fractal

Functions
[m] = mandelbrot (width, iterations)
[outline] = snowflake (n)

```
MANDELBROT Generate Mandelbrot set with WIDTH pixels on the X axis.

   m = MANDELBROT(WIDTH, ITERATIONS) uses the "Escape Time" algorithm to
   generate a Mandelbrot set WIDTH pixels wide and (5/6 * WIDTH) pixels
   tall. The color of each pixel depends on the number of steps required
   for an iterative series to diverge to infinity. If the series at a
   given pixel does not diverge after ITERATIONS steps, the pixel is
   colored black.

   Algorithms based on the Wikipedia article:
   http://en.wikipedia.org/wiki/Mandelbrot_set
```

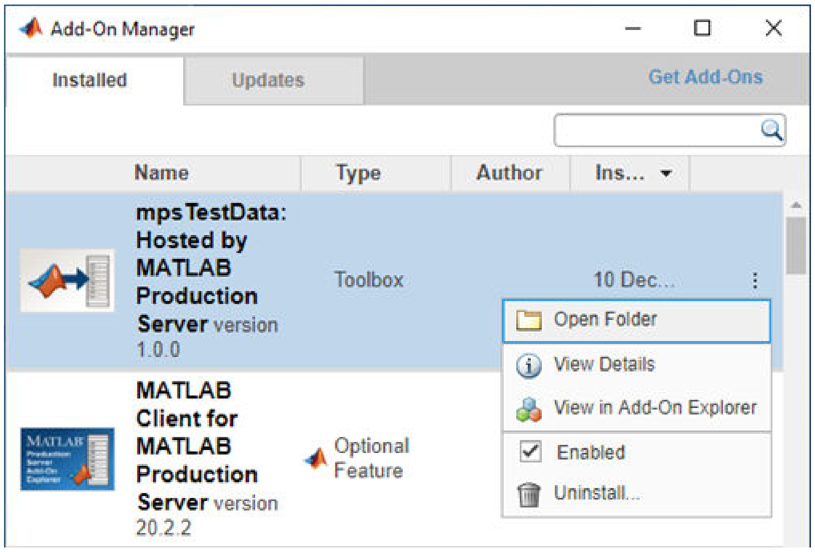
Manage Add-Ons

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 8-14.

For information about specifying a bearer token from the MATLAB command prompt, see `prodserver.addon.accessTokenPolicy`.

See Also

`prodserver.addon.install` | `prodserver.addon.accessTokenPolicy` | `prodserver.addon.Explorer` | `prodserver.addon.availableAddOns` | `prodserver.addon.isInstalled` | `prodserver.addon.uninstall`

More About

- “Execute Deployed MATLAB Functions” on page 8-5

- “Execute Deployed Functions Using HTTPS” on page 8-17
- “Application Access Control” on page 8-14

Deploy Add-Ons

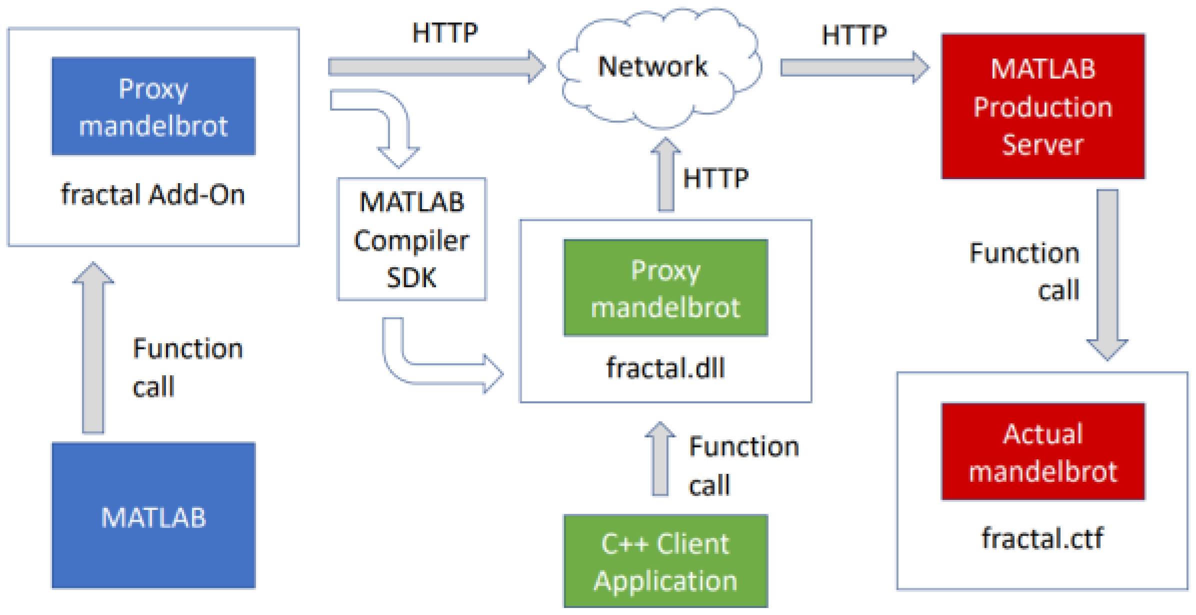
MATLAB deployment tools such as MATLAB Compiler and MATLAB Compiler SDK package MATLAB functions for deployment to environments external to the MATLAB desktop. These deployment tools can also package the proxy functions that MATLAB Production Server add-ons install to create deployable software components 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 `mandlebrot` 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 1-7.
2. Deploy the fractal archive to a MATLAB Production Server instance. For more information about deploying the archive, see "Deploy Archive to MATLAB Production Server".

   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 8-5.

You can verify that the fracatl 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 8-5
- “Execute Deployed Functions Using HTTPS” on page 8-17
- “Configure Client-Server Communication” on page 8-11

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 8-5.

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 8-5.

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 8-5.

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: `'AAAAAAAAAABBBBAAAAAAAMLheAAAAAAA0%2BuSepl%2BULvsea4JtiGRiSDSJSI %3DEUifiRmndf5E2XzMDjRfl76ZC9Ub0wnz4XsNiRVBChTYbJcE3F'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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`

Version History

Introduced in R2020b

See Also

`prodserver.addon.set`

Topics

"Application Access Control" on page 8-14
"Execute MATLAB Functions Using HTTPS"

prodserver.addon.availableAddOns

MATLAB Production Server add-ons available on active server instance

Syntax

```
addons = prodserver.addon.availableAddOns(host,port)
addons = prodserver.addon.availableAddOns(host,port,'TransportLayerSecurity',
tf)
```

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.

`addons = prodserver.addon.availableAddOns(host,port,'TransportLayerSecurity',
tf)` additionally lets you specify the URI scheme (HTTP or HTTPS) of the sever.

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 1-2 and “Deploy Archive to MATLAB Production Server”.

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                    Scheme        Host
    _____________    _________    _______    _________    ______________________________________    _______    __________

      "fractal"      "R2020b"     "1.0.0"      true       "76643195-2ba3-4574-8fd0-084cc51251aa"     "http"    "localhos
```

The output indicates that the `fractal` add-on is available and is also installed on the client machine.

Add-Ons Available on Server that Uses HTTPS

Find the names of add-ons that are available on an active server instance that uses HTTPS.

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 1-2 and "Deploy Archive to MATLAB Production Server".

Start the server instance running at `localhost` and port `57143`.

Find which add-ons are available on the server.

```
addons = prodserver.addon.availableAddOns('localhost',57143,'TransportLayerSecurity',true)
```

```
addons =

  1×8 table

       Name          Release      Version    Installed                   Identifier                    Scheme       Host
    ____________    _________    _______    _________    ______________________________________    _______    _________

     "fractal"      "R2020b"     "1.0.0"      true       "76643195-2ba3-4574-8fd0-084cc51251aa"     "https"    "localhos
```

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`

`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

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.

Version History

Introduced in R2019b

See Also

`prodserver.addon.install` | `prodserver.addon.uninstall` | `prodserver.addon.installFolder` | `prodserver.addon.isInstalled`

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
```

Version History

Introduced in R2019b

See Also

`prodserver.addon.install` | `prodserver.addon.uninstall`

Topics

“Execute Deployed MATLAB Functions” on page 8-5

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-Sever 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: [1×0 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 8-17.

Version History

Introduced in R2020a

See Also

`prodserver.addon.set`

External Websites

“Execute MATLAB Functions Using HTTPS”

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`.

You can install multiple add-ons that have the same name but are hosted on different servers. The proxy functions that the add-ons create appear on the MATLAB search path. When you call a proxy function, the function with the same name that appears nearest to the top of the MATLAB search path is invoked. For more information about the MATLAB search path, see “What Is the MATLAB Search Path?” (MATLAB).

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 1-2 and “Deploy Archive to MATLAB Production Server”.

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                          Endpoint
    __________________  _______    _______    ______________________________________    ________________________

    "fractal (R2020b)"  "1.0.0"     true      "599ce38c-eea1-4011-85b9-8d301b4d5375"    "http://10.2.2.5:57142"
```

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 1-2 and "Deploy Archive to MATLAB Production Server".

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 =

  1×5 table

          Name          Version    Enabled                 Identifier                          Endpoint
    __________________  _______    _______    ______________________________________    ________________________

    "fractal (R2020b)"  "1.0.0"     true      "599ce38c-eea1-4011-85b9-8d301b4d5375"    "http://10.2.2.5:57142"
```

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 1-2 and "Deploy Archive to MATLAB Production Server".

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 =

  1×5 table

          Name          Version    Enabled                 Identifier                          Endpoint
    __________________  _______    _______    ______________________________________    ________________________

    "fractal (R2020b)"  "1.0.0"     true      "599ce38c-eea1-4011-85b9-8d301b4d5375"    "https://10.2.2.5:57143"
```

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 1-2 and “Deploy Archive to MATLAB Production Server”.

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 =

  1×5 table

           Name          Version     Enabled                   Identifier                             Endpoint
    __________________   _______     _______    ______________________________________    ________________________

    "fractal (R2020b)"   "1.0.0"      true      "599ce38c-eea1-4011-85b9-8d301b4d5375"    "http://10.2.2.5:57142"
```

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`

Version History
Introduced in R2019b

See Also
`prodserver.addon.uninstall` | `prodserver.addon.isInstalled` | `prodserver.addon.installFolder` | `prodserver.addon.availableAddOns`

Topics
"Execute Deployed MATLAB Functions" on page 8-5
"Connect MATLAB Session to MATLAB Production Server" on page 8-2

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 MATLAB Production Server"
```

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 MATLAB Production Server(2)'
```

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 MATLAB Production Server"
```

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`

Version History

Introduced in R2020b

See Also

`prodserver.addon.install` | `prodserver.addon.isInstalled` | `prodserver.addon.availableAddOns`

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 1-2 and “Deploy Archive to MATLAB Production Server”.

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.

Version History

Introduced in R2020b

See Also

`prodserver.addon.install` | `prodserver.addon.uninstall` | `prodserver.addon.availableAddOns` | `prodserver.addon.isInstalled`

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` | `unint16`

`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.

Version History
Introduced in R2019b

See Also

`prodserver.addon.install` | `prodserver.addon.isAddOnFcn` | `prodserver.addon.set` | `prodserver.addon.availableAddOns` | `prodserver.addon.installFolder`

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 8-17.

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.

Version History

Introduced in R2020a

See Also

`prodserver.addon.get`

Topics

“Execute MATLAB Functions Using HTTPS”

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`

Version History
Introduced in R2019b

See Also
`prodserver.addon.install`

Streaming Functions

categoryList

Package: `matlab.io.stream.event`

Kafka stream provider property list

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
list = categoryList(ks,cat)
```

Description

`list = categoryList(ks,cat)` returns the names and categories of all the Kafka® stream provider properties and their respective values from the categories `cat` for the object `ks`.

The returned `list` is a cell array of alternating structures and values of the form `{name_sctruct1,value1,...,name_structN,valueN}`, where:

- `name_struct` is a structure with fields `"category"` and `"name"`. These fields define the category name and property name, respectively.
- `value` contains the value of the property named in the preceding structure. `value` can be of any type.

Examples

List Kafka Provider Properties and Values from Specific Category

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Create a `KafkaStream` object connected to the Kafka host and also specify security properties during object creation.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan", ...
                    "security.protocol","SSL", ...
                    "ssl.truststore.type","PEM", ...
                    "ssl.truststore.location","kafka-boston.pem");
```

Get the names, categories, and values of the two Kafka provider properties in the `"Uncategorized"` category.

- Property `ssl.truststore.location` has a value of `"kafka-boston.pem"`
- Property `ssl.truststore.type` has a value of `"PEM"`

```
list = categoryList(ks,"Uncategorized")
prop1 = list{1}
prop2 = list{3}
```

```
list =

  1×4 cell array

    {1×1 struct}    {["kafka-boston.pem"]}    {1×1 struct}    {["PEM"]}


prop1 =

  struct with fields:

        name: 'ssl.truststore.location'
    category: 'Uncategorized'


prop2 =

  struct with fields:

        name: 'ssl.truststore.type'
    category: 'Uncategorized'
```

Input Arguments

ks — Object connected to Kafka stream topic
`KafkaStream` object

Object connected to a Kafka stream topic, specified as a `KafkaStream` object.

cat — Kafka stream provider category name
string scalar | character vector | string array | cell array of character vectors

Kafka stream provider category name, specified as a string scalar, character vector, string array, or cell array of character vectors. This table shows the categories that Streaming Data Framework for MATLAB Production Server supports. Properties can belong to more than one category.

Category	Description
`Uncategorized`	Contains all Kafka provider properties that do not fall into any other category **Kafka Property Example**: `ssl.truststore.pem`
`Consumer`	Contains provider properties specific to a Kafka consumer **Kafka Property Example**: `security.protocol`
`ConsumerTopic`	Contains provider properties specific to reading a topic

Category	**Description**
`Producer`	Contains provider properties specific to a Kafka producer **Kafka Property Example**: `security.protocol`
`ProducerTopic`	Contains provider properties specific to writing to a Kafka topic **Kafka Property Example**: `max.request.size`
`CreateTopic`	Contains provider properties specific to creating a Kafka topic **Kafka Property Example**: `retention.ms`
`KafkaConnector`	Contains provider properties for interacting with the Kafka Connector interface. `readtimetable` uses this interface when you set the `Order` property to `EventTime` in a `KafkaStream` object. **Kafka Property Example**: `sasl.jaas.config`
`librdkafka`	Contains provider properties for interacting with the Kafka interface `librdkafka`. `readtimetable` uses this interface when you set the `Order` property to `IngestTime` in a `KafkaStream` object. `writetimetable` always uses `librdkafka`. **Kafka Property Example**: `sasl.username`

Example: `categoryList(ks,["WriteTopic" "CreateTopic"])` returns any provider properties set in `ks` that are related to creating or writing a topic.

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

Version History

Introduced in R2022b

See Also

`getProviderProperties` | `setProviderProperties` | `isProperty`

Topics

"Connect to Secure Kafka Cluster" on page 11-9

createTopic

Package: `matlab.io.stream.event`

Create topic in Kafka cluster

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
createTopic(ks)
createTopic(ks,MissingTopic=action)
```

Description

`createTopic(ks)` creates a topic on the Kafka host, if Kafka cluster permissions allow creation. The `KafkaStream` object `ks` specifies the topic to create and the network address of the Kafka host that contains the topic.

`createTopic(ks,MissingTopic=action)` specifies whether to create a topic Kafka or fail when the topic is missing. If the Kafka cluster that you are writing to is configured to auto-create topics, specifying `action` has no effect.

Examples

Create Topic in Kafka Cluster

Assume that you have a Kafka server running at network address `kafka.host.com:9092`.

Create an object connected to the Kafka host that interacts with a topic called `CoolingFan`.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan");
```

If the `CoolingFan` topic does not exist during object creation, you can create it. You must have the necessary permissions to create topics.

```
createTopic(ks)
```

Input Arguments

ks — Object connected to Kafka stream topic
`KafkaStream` object

Object connected to a Kafka stream topic, specified as a `KafkaStream` object.

action — Action to take if topic does not exist
`create` (default) | `fail`

Action to take if the topic does not exist, specified as one of the following values:

- `create` — Creates new topic, if you have the required permissions on the Kafka host.
- `fail` — Does not create a new topic.

Data Types: `char` | `string`

Version History

Introduced in R2022b

See Also

`readtimetable` | `writetimetable` | `detectImportOptions` | `deleteTopic` | `kafkaStream`

deleteTopic

Package: `matlab.io.stream.event`

Remove topic from Kafka cluster

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
deleteTopic(ks)
```

Description

`deleteTopic(ks)` removes a topic from a Kafka cluster, if Kafka permissions allow deletion. The `KafkaStream` object `ks` specifies the topic to delete and the network address of the Kafka host that contains the topic. This deletion is permanent. The deleted topic and the data it contains cannot be recovered.

Examples

Delete Kafka Topic

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Create an object connected to the `CoolingFan` topic.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan");
```

If you no longer require the topic and the data in it, you can delete it. You must have the necessary permissions to delete topics.

```
deleteTopic(ks)
```

Input Arguments

ks — Object connected to Kafka stream topic
`KafkaStream` object

Object connected to a Kafka stream topic, specified as a `KafkaStream` object.

Version History

Introduced in R2022b

See Also

`createTopic` | `readtimetable` | `detectImportOptions` | `kafkaStream`

detectExportOptions

Package: `matlab.io.stream.event`

Create export options based on event stream content

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
opts = detectExportOptions(stream,row)
opts = detectExportOptions(stream,row,Format=format)
```

Description

`opts = detectExportOptions(stream,row)` detects export options from an event stream and returns them in `opts`. These options specify the rules for transforming MATLAB variables into streaming data. To determine which streaming data type best matches the type of each column variable, `detectExportOptions` examines a row of timetable data and includes the type information in `opts`.

To get and set the types of the variables as they are exported from MATLAB to the stream, use `getvartype` and `setvartype`.

To export data from MATLAB to a stream, use `writetimetable`.

`opts = detectExportOptions(stream,row,Format=format)` optionally sets the output stream format.

Examples

Detect Data Export Options from Event Stream

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `Triangles`.

Create a `KafkaStream` object connected to the `Triangles` topic.

```
inKS = kafkaStream("kafka.host.com",9092,"Triangles");
```

Read events from the `Triangles` topic into a timetable. Preview the data by viewing the first row. The `a`, `b`, and `c` triangle side lengths are stored as strings.

```
tt = readtimetable(inKS);
row = tt(1,:)
```

```
row =

  1×3 timetable
```

```
     timestamp      a       b       c
    ___________    ____    ____    ____

    03-Sep-2022    "15"    "31"    "36"
```

Use `detectExportOptions` to generate an `ExportOptions` object from the Kafka stream object. The function obtains the types used to export the variables from the first row of the timetable.

```
opts = detectExportOptions(inKS,row);
```

Use `getvartype` to confirm that the side length variables are currently exported to the stream as strings.

```
type = getvartype(opts,["a" "b" "c"]);
```

```
type = 

  1×3 string array

    "string"    "string"    "string"
```

Update the export options so that the side lengths are exported as `double` values. Confirm the updated options by using `getvartype`.

```
opts = setvartype(opts,["a","b","c"],"double");

[name,type] = getvartype(opts);
fprintf("%s: %s\n", [name; type])
```

```
a: double
b: double
c: double
```

Connect to the stream to export data to `numericTriangles`.

```
outKS = kafkaStream("kafka.host.com",9092,"numericTriangles", ...
    ExportOptions=opts)
```

```
outKS = 

  KafkaStream with properties:

                  Topic: "numericTriangles"
                  Group: "85c42e39-695d-467a-86f0-f0095792e7de"
                  Order: EventTime
                   Host: "kafka.host.com"
                   Port: 9092
      ConnectionTimeout: 30
         RequestTimeout: 61
          ImportOptions: "None"
          ExportOptions: "Source: string"
          PublishSchema: "true"
             WindowSize: 50
            KeyVariable: "key"
            KeyEncoding: "utf16"
                KeyType: "text"
           KeyByteOrder: "BigEndian"
           BodyEncoding: "utf8"
             BodyFormat: "JSON"
```

```
              ReadLimit: "Size"
    TimestampResolution: "Milliseconds"
```

Export the timetable to the new stream. The triangle side lengths in this stream are of type `double`.

```
writetimetable(outKS,tt);
```

Input Arguments

`stream` — Object connected to event stream
`KafkaStream` object | `TestStream` object

Object connected to an event stream, specified as a `KafkaStream` or `TestStream` object.

`row` — Row of MATLAB timetable data
timetable

Row of MATLAB timetable data. `row` must be an example of the data that you intend to write to the stream. `detectExportOptions` uses the columns of this row to set the variable names and types in the export options.

Example: `tt(1,:)` extracts the first row of data from timetable `tt`.

`format` — Format of output stream
`"Event"` (default) | `"InfluxDB"`

Format of the output stream, specified as one of these options:

- `"Event"` — Native, default even stream format
- `"InfluxDB"` — Format specifically used by InfluxDB

Data Types: `char` | `string`

Output Arguments

`opts` — Event stream export options
`ExportOptions` object

Event stream export options, returned as an `ExportOptions` object.

Version History

Introduced in R2022b

See Also

`ExportOptions` | `detectImportOptions` | `setvartype` | `getvartype`

detectImportOptions

Package: `matlab.io.stream.event`

Create import options based on event stream content

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
opts = detectImportOptions(stream)
opts = detectImportOptions(stream,Event=e)
```

Description

`opts = detectImportOptions(stream)` obtains the event schema from an event stream and based on that schema, returns data import options in the `ImportOptions` object `opts`. The schema includes the names and data types of variables in the event body.

You can modify `opts` later and use it with `readtimetable` to control how MATLAB imports stream events into MATLAB timetables.

`opts = detectImportOptions(stream,Event=e)` analyzes the event structure array `e` to determine the event schema. Use this syntax only if the event stream that `stream` connects to is unavailable. You can either manually create this array, which can be complex, or use one that was previously obtained from the stream, such as from a call to the `readevents` function.

Examples

Detect Import Options from Event Stream

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `Triangles`. Each event has one data column, `"triangle"`, which contains a structure with three fields, `"x"`, `"y"`, `"z"`, containing the integer side lengths of a triangle.

Create a `KafkaStream` object connected to the `Triangles` topic.

```
ks = kafkaStream("kafka.host.com",9092,"Triangles");
```

Create an `ImportOptions` object from the Kafka stream object. The data type of the length of each side is `string`.

```
opts = detectImportOptions(ks)
```

```
opts = 

  ImportOptions with properties:

           VariableNames: ["triangle/x"    "triangle/y"    "triangle/z"]
           VariableTypes: ["string"    "string"    "string"]
             KeyVariable: "key"
   SelectedVariableNames: ["triangle/x"    "triangle/y"    "triangle/z"]
```

To perform mathematical operations on the imported data, update the data type of variables to `double`. Because the side length variables are nested within `"triangle"`, use a forward slash (`"/"`) to specify the path to these variables.

```
opts = setvartype(opts, ["triangle/x", "triangle/y", "triangle/z"], "double")
```

```
opts = 

  ImportOptions with properties:

            VariableNames: ["triangle/x"    "triangle/y"    "triangle/z"]
            VariableTypes: ["double"    "double"    "double"]
              KeyVariable: "key"
    SelectedVariableNames: ["triangle/x"    "triangle/y"    "triangle/z"]
```

Update the `ImportOptions` property of the `KafkaStream` object.

```
ks.ImportOptions = opts
```

```
ks = 

  KafkaStream with properties:

                  Topic: "Triangles"
                  Group: "85c42e39-695d-467a-86f0-f0095792e7de"
                  Order: EventTime
                   Host: "kafka.host.com"
                   Port: 9092
      ConnectionTimeout: 30
         RequestTimeout: 61
          ImportOptions: "Import to MATLAB types"
          ExportOptions: "Source: function eventSchema"
          PublishSchema: "true"
             WindowSize: 50
            KeyVariable: "key"
            KeyEncoding: "utf16"
                KeyType: "text"
           KeyByteOrder: "BigEndian"
           BodyEncoding: "utf8"
             BodyFormat: "JSON"
              ReadLimit: "Size"
    TimestampResolution: "Milliseconds"
```

When importing the triangles, `readtimetable` converts the side lengths to `double` values.

```
tt = readtimetable(ks);
tt(1,:).triangle
```

```
ans = 

  struct with fields:

    x: 3
    y: 4
    z: 5
```

Input Arguments

`stream` — Object connected to event stream

`KafkaStream` object | `TestStream` object

Object connected to an event stream, specified as a `KafkaStream` or `TestStream` object.

e — Event information
structure array

Event information, specified as a structure array. This array is in the format returned by the `readevents` function.

Each structure in the array has these fields.

key — Event key
string array | positive integer

Event key as stored in Kafka, returned as a string array or integer. The key identifies the event source.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`value` — Event value
byte array

Event value, specified as a byte array with a format and encoding determined by the `BodyFormat` and `BodyEncoding` properties of the stream object. The event value does not undergo schema processing and appears exactly as is stored in Kafka, for example, as a JSON string.

Data Types: `string` | `uint8` | `uint16`

`timestamp` — Event timestamp or ingest timestamp
datetime scalar

Timestamp of event occurrence or timestamp of event ingestion in Kafka, specified as a datetime scalar.

Data Types: `datetime`

Output Arguments

`opts` — Event stream import options
`ImportOptions` object

Event stream import options, returned as an `ImportOptions` object.

Version History

Introduced in R2022b

See Also

`ImportOptions` | `eventStreamImportOptions` | `kafkaStream` | `testStream` | `setvartype` | `detectExportOptions`

eventStreamImportOptions

Create options for importing events from stream into MATLAB

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
opts = eventStreamImportOptions(VariableNames=names,VariableTypes=types)
opts = eventStreamImportOptions(KeyVariable=kv)
opts = eventStreamImportOptions(VariableNames=names,VariableTypes=
types,KeyVariable=kv)
```

Description

Use `eventStreamImportOptions` only when you are unable to detect import options from the stream object using `detectImportOptions`.

`opts = eventStreamImportOptions(VariableNames=names,VariableTypes=types)` sets the `VariableNames` and `VariableTypes` properties of the `ImportOptions` object `opts`. `names` and `types` are the names and data types of event variables that you want to import from an event stream into MATLAB.

`opts = eventStreamImportOptions(KeyVariable=kv)` sets the `KeyVariable` property of the `ImportOptions` object `opts`.

`opts = eventStreamImportOptions(VariableNames=names,VariableTypes= types,KeyVariable=kv)` sets the `VariableNames`, `VariableTypes`, and`KeyVariable` properties.

Examples

Create Variable Import Options for Event Stream Data

Create a schema for importing data from an event stream into MATLAB by specifying variable names and their data types to use during the import.

```
names = ["x","symbol"];
types = ["double","string"];
```

Construct an `ImportOptions` object using this data import schema.

```
opts = eventStreamImportOptions(VariableNames=names,VariableTypes=types)
```

```
opts = 

  ImportOptions with properties:

            VariableNames: ["x"    "symbol"]
            VariableTypes: ["double"    "string"]
```

```
              KeyVariable: [0×0 string]
    SelectedVariableNames: ["x"    "symbol"]
```

Apply the import options when creating a `KafkaStream` object.

```
ks = kafkaStream("kafka.host.com",9092,"Your_Kafka_Topic",ImportOptions=opts);
```

Import the data. The `"Your_Kafka_Topic"` topic must have events with exactly two variables, `x` and `symbol`. In addition, the types of these variables must be convertible to double and string, respectively. Otherwise, `readtimetable` throws an error.

```
tt = readtimetable(ks);
```

Input Arguments

names — Variable names
string scalar | string array | cell array of character vectors

Variable names to use when importing variables from the event stream into a timetable, specified as a string scalar, string array, or cell array of character vectors.

Data Types: `string` | `cell`

types — Data type of variables
string scalar | string array | cell array of character vectors

Data type of variables to use when importing variables from the event stream into a timetable, specified as a string scalar, string array, or cell array of character vectors containing a set of valid data type names. The `VariableTypes` property designates the data types.

Data Types: `string` | `cell`

kv — Event key variable name
string scalar | character vector

Event key variable name to use when importing variables from the event stream into a timetable, specified as a string scalar or character vector.

Data Types: `string` | `char`

Version History

Introduced in R2022b

See Also

`detectImportOptions` | `setvartype`

eventStreamProcessor

Apply stream analytic function to event stream

Note This object requires Streaming Data Framework for MATLAB® Production Server™.

Description

Use an `EventStreamProcessor` object to apply a stream analytic function to an event stream. Using `EventStreamProcessor` object functions, you can automatically direct events from an event stream to a streaming analytic function, enabling you to process large amounts of data in event streams.

You can run the streaming analytic function on a known number of event windows synchronously, similar to a `for`-loop. You can also run it with a desktop-hosted server to simulate asynchronous deployment in a production environment.

`EventStreamProcessor` functions can process streaming data sequentially in batches by collecting events into windows of configurable size. When a window is full of the requested number of events, the window of events is passed to the stream processing analytic function. You can then save any results that the analytic function produces and optionally publish them to a different stream.

A stream processing function can be stateful or stateless. For stateful functions, the `EventStreamProcessor` object maintains state between calls to the stream processing function. If the stream processing function changes the state, the function can return the state as a second output argument. The `EventStreamProcessor` object preserves these changes for the next function iteration.

Creation

Syntax

```
esp = eventStreamProcessor(inputStream,streamFcn)
esp = eventStreamProcessor(inputStream,streamFcn,initialState)
esp = eventStreamProcessor( ___ ,Name=Value)
```

Description

`esp = eventStreamProcessor(inputStream,streamFcn)` creates an `EventStreamProcessor` object, which applies the stream function `streamFcn` to the event stream `inputStream`, and sets the `InputStream` and `StreamFunction` properties, respectively, of this object.

`esp = eventStreamProcessor(inputStream,streamFcn,initialState)` creates an `EventStreamProcessor` object that additionally initializes persistent state with the function `initialState` and sets the `InitialState` property. If `streamFcn` is stateful, then `initialState` is required.

`esp = eventStreamProcessor( ___ ,Name=Value)` sets object properties using one or more name-value arguments. Name is a property name on page 10-18 and Value is the corresponding value. You can specify multiple name-value arguments in any order as `Name1=Value1`,...,`NameN=ValueN`.

Properties

`ArchiveName` — Name of generated deployable archive
string array

Name of the deployable archive generated by the `package` function, specified as a string. The default archive name is the name of the streaming function.

Data Types: `string`

`GroupVariable` — Name of event variable used to group events
string array | character vector

Name of the event variable used to group events, specified as a string array or character vector.

If `GroupVariable` is nonempty, each event window is split into groups in which the event variables have the same value. Each group is then sent to the streaming function separately. `GroupVariable` is often set to the event key so that events from each event source are processed independently.

Data Types: `string` | `char`

`InitialState` — Function that creates initial state for streaming analytic function
function handle

Function that creates the initial state for the streaming analytic function, specified as a function handle. If the streaming analytic function is stateful, this property must be set when you create the object.

`InputStream` — Event stream from which the streaming analytic function reads events
`KafkaStream` object | `InMemoryStream` object | `TestStream` object

Event stream from which the streaming analytic function reads events, specified as a `KafkaStream`, `InMemoryStream`, or `TestStream` object.

`OutputStream` — Event stream to which streaming analytic function writes events
`InMemoryStream` object (default) | `KafkaStream` object | `TestStream` object

Event stream to which the streaming analytic function writes events, specified as a `KafkaStream`, `InMemoryStream`, or `TestStream` object.

Note If you are packaging your stream processing function into a deployable archive using the `package` function, do not leave `OutputStream` set to an `InMemoryStream` object. This object is not supported by the `package` function as an output stream.

`StreamFunction` — Streaming analytic function
function handle

Streaming analytic function, specified as a function handle.

Data Types: `function handle`

`ReadPosition` — Position in event stream to read from
`"Beginning"` | `"End"` | `"Current"`

Position in an event stream to read from, specified as one of these values:

- `"Beginning"` — First event available in stream
- `"End"` — Just past the last event in the stream
- `"Current"` — Just past the current event in the stream

Data Types: `string`

`ResetStateOnSeek` — Flag to clear persistent state after calling seek
true (default) | false

Flag to clear persistent state after calling the `seek` function, specified as a logical scalar.

Data Types: `logical`

Object Functions

execute	Execute event stream processing function on specific number of event windows
package	Package stream processing function into deployable archive configured by EventStreamProcessor
seek	Set position in event stream to begin processing events
start	Start processing event streams using local test server
startServer	Start local test server
stop	Stop processing event streams using local test server
stopServer	Shut down local test server

Examples

Iterate Streaming Analytic Function Over Several Event Windows

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `RecamanSequence`.

Create an object connected to the `RecamanSequence` topic.

```
ks = kafkaStream("kafka.host.com",9092,"RecamanSequence");
```

Assume that you have a streaming analytic function `recamanSum` and a function to initialize persistent state called `initRecamanSum`.

Create an `EventStreamProcessor` object to run the `recamanSum` function and initializes persistent state with the `initRecamanSum` function.

```
esp = eventStreamProcessor(ks,@recamanSum,@initRecamanSum);
```

```
esp = 

  EventStreamProcessor with properties:
      StreamFunction: @recamanSum
         InputStream: [1×1 matlab.io.stream.event.KafkaStream]
        OutputStream: [1×1 matlab.io.stream.event.InMemoryStream]
        InitialState: @initRecamanSum
       GroupVariable: [0×0 string]
        ReadPosition: Beginning
         ArchiveName: "recamanSum"
    ResetStateOnSeek: 1
```

Iterate the streaming analytic function over the stream ten times.

```
execute(esp,10);
```

Examine the results.

```
result = readtimetable(esp.OutputStream)
```

Simulate Production Using Test Server

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `RecamanSequence`.

Also assume that you have a streaming analytic function `recamanSum` and a function `initRecamanSum` to initialize persistent state.

Create a `KafkaStream` object connected to the `RecamanSequence` topic.

```
ks = kafkaStream("kafka.host.com",9092,"RecamanSequence");
```

Create another `KafkaStream` object to write the results of the streaming analytic function to a different topic called `RecamanSequenceResults`.

```
outKS = kafkaStream("kafka.host.com",9092,"RecamanSequenceResults");
```

Create an `EventStreamProcessor` object that runs the `recamanSum` function and initializes persistent state with the `initRecamanSum` function.

```
esp = eventStreamProcessor(ks,@recamanSum,@initRecamanSum,OutputStream=outKS);
```

```
esp = 

  EventStreamProcessor with properties:

      StreamFunction: @recamanSum
         InputStream: [1×1 matlab.io.stream.event.KafkaStream]
        OutputStream: [1×1 matlab.io.stream.event.KafkaStream]
        InitialState: @initRecamanSum
       GroupVariable: [0×0 string]
        ReadPosition: Beginning
         ArchiveName: "recamanSum"
    ResetStateOnSeek: 1
```

Using the MATLAB editor, you can set breakpoints in the `recamanSum` function to examine the incoming streaming data when you start the server.

Start the test server.

Note To use the test server, you require MATLAB Compiler SDK.

```
startServer(esp);
```

Doing so opens the **Production Server Compiler** app. When the app opens, you must start the server manually.

To start the test server from the app, click **Test Client**, and then click **Start**. For an example on how to use the app, see “Test Client Data Integration Against MATLAB” (MATLAB Compiler SDK).

Navigate back to the MATLAB command prompt to start processing events.

```
start(esp);
```

In the **Production Server Compiler** app, the test server receives data.

From the MATLAB editor, if you set breakpoints, you can use the debugger to examine the data, state, and results of the function processing. Click **Continue** to continue debugging or **Stop** when you finish debugging.

From the MATLAB command prompt, stop the server.

```
stop(esp);
```

Read results from the output stream.

```
results = readtimetable(outKS);
```

Version History

Introduced in R2022b

See Also

`seek` | `streamingDataCompiler` | `execute` | `package` | `seek` | `start` | `startServer` | `stop` | `stopServer`

Topics

“Test Streaming Analytic Function Using Local Test Server” on page 11-12
“Deploy Streaming Analytic Function to MATLAB Production Server” on page 11-17

execute

Package: `matlab.io.stream.event`

Execute event stream processing function on specific number of event windows

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
execute(esp,n)
```

Description

`execute(esp,n)` runs the streaming processing function specified in the `Name` property of processing object `esp` synchronously on `n` event windows.

The `execute` function starts processing event windows at the current read position of the stream. Each event window is adjacent to the previous window, with no gaps between windows. To change the starting position of the entire sequence, call `seek` before calling `execute`.

The first call to `execute` reads events from the position in the data stream where the read position was when `esp` was constructed. On subsequent calls to `execute`, the read position is set to `Current`. To change this behavior, call `seek` before `execute`.

Examples

Execute Event Stream Processing Function

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `RecamanSequence`.

Create an object connected to the `RecamanSequence` topic.

```
ks = kafkaStream("kafka.host.com", 9092, "RecamanSequence");
```

Assume that you have a streaming analytic function `recamanSum` and a function to initialize persistent state called `initRecamanSum`.

Create an `EventStreamProcessor` object that runs the `recamanSum` function and initializes persistent state with the `initRecamanSum` function.

```
esp = eventStreamProcessor(ks,@recamanSum, @initRecamanSum);
```

```
esp = 

  EventStreamProcessor with properties:

      StreamFunction: @recamanSum
         InputStream: [1×1 matlab.io.stream.event.KafkaStream]
```

```
        OutputStream: [1×1 matlab.io.stream.event.InMemoryStream]
        InitialState: @initRecamanSum
       GroupVariable: [0×0 string]
        ReadPosition: Beginning
         ArchiveName: "recamanSum"
    ResetStateOnSeek: 1
```

Iterate over the streaming analytic function ten times.

```
execute(esp,10);
```

Move the read position indicator to the beginning of the default output data stream.

```
seek(esp.OutputStream,"Beginning");
```

Examine the results.

```
result = readtimetable(esp.OutputStream)
```

Input Arguments

esp — Object to process event streams
`EventStreamProcessor` object

Object to process event streams, specified as an `EventStreamProcessor` object.

n — Number of event windows
positive integer

Number of event windows, specified as a positive integer.

Version History

Introduced in R2022b

See Also

`eventStreamProcessor` | `package` | `seek` | `start` | `startServer` | `stop` | `stopServer`

Topics
"Process Kafka Events Using MATLAB" on page 11-5

ExportOptions

Export options for event stream

Note This object requires Streaming Data Framework for MATLAB® Production Server™.

Description

An `ExportOptions` object specifies how MATLAB exports data from timetables to external event streams, such as Kafka streams. The object contains properties that control the data export process, including the transformation of event data to the specified type.

Creation

Create an `ExportOptions` object by using the `detectExportOptions` function. This function detects and populates the export rules based on the configuration of the event stream specified by `stream` and the variables in the timetable row specified by `row`.

```
opts = detectExportOptions(stream,row)
```

After creating `opts`, use `setvartype` to change the types of the variables as they are exported from MATLAB to the stream. To export stream data, use the `writetimetable` function.

Properties

`Scope` — Scope of schema
`"None"` | `"Event"` | `"Window"` | `"Stream"`

Scope of the schema, specified as one of these options:

- `"None"` — Schema has no scope.
- `"Event"` — Schema applies to a single event.
- `"Window"` — Schema applies to all events in the window.
- `"Stream"` — Schema applies to all windows in the stream.

Use `Scope` to optimize how often to refresh the schema.

Data Types: `string` | `char`

`Schema` — Schema describing variables being exported
JSON-formatted text | function handle | `Schema` object

Schema describing the MATLAB variables being exported to the event stream, specified as one of these values:

- JSON-formatted text (string or character vector)
- Function handle that evaluates to JSON-formatted text

- `Schema` object that produces JSON-formatted text

This schema defines the rules for converting variables to the appropriate data type.

Example: `'[{"name":"N","type":"int64","size":[1,1],"missingValue":0,"categorical":false},{"name":"X","type":"double","size":[1,1],"missingValue":"","categorical":false}]'` describes a schema with two variables: `N` (`int64`) and `X` (`double`).

Data Types: `char` | `string` | `function_handle`

`OutputLocation` — Location of schema
`"Event"` | `"Window"`

Location of the schema, specified as one of these options:

- `"Event"` — Schema is embedded in a single event.
- `"Window"` — Schema is embedded in an event window.

Data Types: `char` | `string`

`Format` — Structure of text representation of schema
`"Event"` | `"InfluxDB"`

Structure of the text representation of the schema, specified as one of these options:

- `"Event"` — Native, default even stream format
- `"InfluxDB"` — Format specifically used by InfluxDB

Data Types: `char` | `string`

`Content` — Content of schema stored in output location
`"None"` | `"Literal"` | `"GeneratorFunction"` | `"RegistryID"`

Content of the schema as it is stored in `OutputLocation`, specified as one of these options:

- `"None"` — No schema specified
- `"Literal"` — The literal schema (a `struct` or JSON text)
- `"GeneratorFunction"` — A function handle that generates the schema
- `"RegistryID"` — A string that identifies the schema

Data Types: `char` | `string`

Object Functions

setvartype	Set data types used to import and export variables to stream

Examples

Create Export Options from Event Stream Data

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has the topics `Triangles` and `numericTriangles`.

Create a `KafkaStream` object connected to the `Triangles` topic.

```
inKS = kafkaStream("kafka.host.com",9092,"Triangles");
```

Read events from the `Triangles` topic into a timetable. Preview the data by viewing the first row. The `a`, `b`, and `c` triangle side lengths are stored as strings.

```
tt = readtimetable(inKS);
row = tt(1,:)
```

```
row =

  1×3 timetable

     timestamp        a       b       c  
    ___________     ____    ____    ____

    03-Sep-2022     "15"    "31"    "36"
```

Use `detectExportOptions` to generate an `ExportOptions` object from the Kafka stream object. The function obtains the types used to export the variables from the first row of the timetable.

```
opts = detectExportOptions(inKS,row);
```

Use `getvartype` to confirm that the side length variables are currently exported to the stream as strings.

```
type = getvartype(opts,["a" "b" "c"]);
```

```
type =

  1×3 string array

    "string"    "string"    "string"
```

Update the export options so that the side lengths are exported as `double` values. Confirm the updated options by using `getvartype`.

```
opts = setvartype(opts,["a","b","c"],"double");

[name,type] = getvartype(opts);
fprintf("%s: %s\n", [name; type])
```

```
a: double
b: double
c: double
```

Connect to the stream to export data to `numericTriangles`.

```
outKS = kafkaStream("kafka.host.com",9092,"numericTriangles", ...
    ExportOptions=opts)
```

```
outKS = 

  KafkaStream with properties:

                  Topic: "numericTriangles"
                  Group: "85c42e39-695d-467a-86f0-f0095792e7de"
                  Order: EventTime
                  Host: "kafka.host.com"
                  Port: 9092
     ConnectionTimeout: 30
        RequestTimeout: 61
         ImportOptions: "None"
         ExportOptions: "Source: string"
         PublishSchema: "true"
            WindowSize: 50
           KeyVariable: "key"
           KeyEncoding: "utf16"
               KeyType: "text"
          KeyByteOrder: "BigEndian"
          BodyEncoding: "utf8"
            BodyFormat: "JSON"
             ReadLimit: "Size"
   TimestampResolution: "Milliseconds"
```

Export the timetable to the new stream. The triangle side lengths in this stream are of type `double`.

```
writetimetable(outKS,tt);
```

Version History

Introduced in R2022b

See Also

`ImportOptions` | `detectExportOptions` | `writetimetable`

flush

Package: `matlab.io.stream.event`

Reset read window boundaries

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
flush(ks)
```

Description

`flush(ks)` resets the read window boundaries to enable reading of incomplete windows of data from the Kafka stream `ks`.

Examples

Read Incomplete Window of Data

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Also assume that the `CoolingFan` topic has 5120 messages.

Create a `KafkaStream` object to connect to the Kafka host. Configure the object to read 500 messages, or events, at a time.

```
inKS = kafkaStream("kafka.host.com",9092,"CoolingFan",Rows=500);
```

Configure a second `KafkaStream` object to connect to the `CoolingFanOut` topic. The Kafka host writes messages to this topic.

```
outKS = kafkaStream("kafka.host.com",9092,"CoolingFanOut",Rows=500);
```

Read 10 windows of message data from `CoolingFan` and write the data to `CoolingFanOut`. This operation processes the first 5000 messages from `CoolingFan`.

```
for idx = 1:10
    tt = readtimetable(inKS);
    writetimetable(outKS,tt)
end
```

Read one more window of messages from `CoolingFan`. This topic has only 120 messages left to be read. Because `readtimetable` does not receive a full window of 500 messages, it times out and does not read the remaining messages.

```
tt = readtimetable(inKS)
tt =

  0×0 empty timetable
```

To enable reading the 120 remaining messages, flush the stream.

```
flush(inKS)
```

Read the remaining messages into a timetable and write them to `CoolingFanOut`. Though the number of messages is smaller than the window size, `readtimetable` is able to read the remaining messages.

```
tt = readtimetable(inKS);
writetimetable(outKS,tt)
```

Input Arguments

ks — Object connected to Kafka stream topic
`KafkaStream` object

Object connected to a Kafka stream topic, specified as a `KafkaStream` object.

Version History

Introduced in R2022b

See Also

`readtimetable` | `seek` | `writetimetable`

getProviderProperties

Package: `matlab.io.stream.event`

Kafka stream configuration property data

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
prop = getProviderProperties(ks)
prop = getProviderProperties(ks,name)
prop = getProviderProperties( ___ ,Category=cat)
[prop,val] = getProviderProperties( ___ )
```

Description

`prop = getProviderProperties(ks)` returns the names and categories of the Kafka stream provider properties on page 10-33 in the structure array `prop`. The returned property names and categories are the ones specified during the creation of the Kafka stream connector object `ks`.

`prop = getProviderProperties(ks,name)` returns only the properties with the provider property names specified by `name`.

`prop = getProviderProperties( ___ ,Category=cat)` returns only the properties that belong to the provider category `cat`, using either of the preceding syntaxes.

`[prop,val] = getProviderProperties( ___ )` also returns a cell array of the values for each returned property.

Examples

Get Provider Property Data

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Create a `KafkaStream` object connected to the Kafka host and also specify Kafka provider properties during object creation.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan", ...
"security.protocol","SSL","ssl.truststore.type","PEM", ...
"ssl.truststore.location","kafka-boston.pem","retention.ms",500);
```

Get the names and categories for all provider properties.

```
prop = getProviderProperties(ks)
```

```
prop =
  8×1 struct array with fields:

    name
    category
```

Display the property categories and names.

```
string({prop.category})' + "/" + string({prop.name})'
```

```
ans = 

  8×1 string array

    "Consumer/auto.offset.reset"
    "Consumer/security.protocol"
    "CreateTopic/retention.ms"
    "KafkaConnector/sasl.jaas.config"
    "Producer/security.protocol"
    "Uncategorized/ssl.truststore.location"
    "Uncategorized/ssl.truststore.type"
    "librdkafka/sasl.username"
```

Get Provider Properties by Name

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Create a `KafkaStream` object connected to the Kafka host and also specify Kafka provider properties during object creation.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan", ...
"security.protocol","SSL","ssl.truststore.type","PEM", ...
"ssl.truststore.location","kafka-boston.pem","retention.ms",500);
```

Get the names and categories for two properties. Because the `max.poll.records` property is not set in `ks`, the `getProviderProperties` function does not return data for that property.

```
prop = getProviderProperties(ks,["max.poll.records" "retention.ms"])
```

```
prop = 

  struct with fields:

        name: 'retention.ms'
    category: 'CreateTopic'
```

Get Provider Properties by Category

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Create a `KafkaStream` object connected to the Kafka host and also specify Kafka provider properties during object creation.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan", ...
"security.protocol","SSL","ssl.truststore.type","PEM", ...
"ssl.truststore.location","kafka-boston.pem","retention.ms",500);
```

Get data for the properties that belong to the `CreateTopic` category. This category includes only one property.

```
prop = getProviderProperties(ks,Category="CreateTopic")
```

```
prop = 

  struct with fields:

        name: 'retention.ms'
    category: 'CreateTopic'
```

Get Provider Property Names and Values

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Create a `KafkaStream` object connected to the Kafka host and also specify Kafka provider properties during object creation.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan", ...
"security.protocol","SSL","ssl.truststore.type","PEM", ...
"ssl.truststore.location","kafka-boston.pem","retention.ms",500);
```

Display the categories, names, and values. The `sasl.jaas.config` property value fails to synthesize because `ks` is missing a dependent property. Instead of returning a value for this property, `getProviderProperties` returns an `MException` object describing this error.

```
[{"Category" "Name" "Value"};{prop.category}' {prop.name}' val']
```

```
ans =

  3×3 cell array

    {["Category"    ]}    {["Name"            ]}    {["Value"     ]}
    {'KafkaConnector'}    {'sasl.jaas.config' }    {1×1 MException}
    {'Producer'      }    {'security.protocol'}    {["SSL"       ]}
```

Input Arguments

ks — Object connected to Kafka stream topic
`KafkaStream` object

Object connected to a Kafka stream topic, specified as a `KafkaStream` object.

name — Kafka stream provider property names
string scalar | character vector | string array | cell array of character vectors

Kafka stream provider property names, specified as a string scalar, character vector, string array, or cell array of character vectors. If a property is not set in `ks`, than `getProviderProperties` does not return a name and category for that property in `prop` or an optional corresponding value in `val`.

Example: `[prop, val] = getProviderProperties(ks,"retention.ms")` returns data for the `retention.ms` property.

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

`cat` — Kafka stream provider category names
string scalar | character vector | string array | cell array of character vectors

Kafka stream provider category names, specified as a string scalar, character vector, string array, or cell array of character vectors.

The specified categories must be present in `ks`. To get a list of valid categories, use the `ks.PropertyCategories` property.

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

Output Arguments

`prop` — Kafka stream provider property names and categories
structure array

Kafka stream provider property names and categories, returned as a structure array. Each structure corresponds to a provider property in `ks` and has these fields:

- `name` — Provider property name, returned as a string
- `category` — Category that the provider property belongs to, returned as a string

Because properties can belong to more than one category, the category and name uniquely identity a property.

`val` — Kafka stream provider property values
cell array

Kafka stream provider property values, returned as a cell array of values for the property returned in `prop`. If `ks` is missing a dependent property needed to derive a property value, then in place of that value, `getProviderProperties` returns an `MException` object that describes the error.

More About

Stream Provider Properties

Stream providers such as Kafka have vendor-specific configuration properties that influence how the stream provider operates on the stream. Some properties apply to a single type of operation, such as topic creation, while other properties apply to multiple operations. Properties are grouped into categories so that properties are sent only to the provider with operations that support them. Because the same property can belong to multiple categories, each property is uniquely identified by the combination of its name and category.

Version History
Introduced in R2022b

See Also

`setProviderProperties` | `categoryList` | `isProperty`

Topics

"Connect to Secure Kafka Cluster" on page 11-9

getvartype

Package: `matlab.io.stream.event`

Data types used to export variables to stream

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
type = getvartype(opts)
[name,type] = getvartype(opts)
___ = getvartype(opts,selection)
```

Description

`type = getvartype(opts)` returns the data types of variables when they are exported to an event stream using the export options specified by `opts`.

`[name,type] = getvartype(opts)` also returns the names of the variables.

`___ = getvartype(opts,selection)` returns the data types, and optionally the names, only for the variables specified by `selection`.

Examples

Get Data Types of Exported Variables in Event Stream

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has the topics `Triangles` and `numericTriangles`.

Create a `KafkaStream` object connected to the `Triangles` topic.

```
inKS = kafkaStream("kafka.host.com",9092,"Triangles");
```

Read events from the `Triangles` topic into a timetable. Preview the data by viewing the first row. The `a`, `b`, and `c` triangle side lengths are stored as strings.

```
tt = readtimetable(inKS);
row = tt(1,:)
```

```
row =

  1×3 timetable

     timestamp      a       b       c  
    ___________    ____    ____    ____

    03-Sep-2022    "15"    "31"    "36"
```

Use `detectExportOptions` to generate an `ExportOptions` object from the Kafka stream object. The function obtains the types used to export the variables from the first row of the timetable.

```
opts = detectExportOptions(inKS,row);
```

Use `getvartype` to confirm that the side length variables are currently exported to the stream as strings.

```
type = getvartype(opts,["a" "b" "c"]);
```

```
type = 

  1×3 string array

    "string"    "string"    "string"
```

Update the export options so that the side lengths are exported as `double` values. Confirm the updated options by using `getvartype`.

```
opts = setvartype(opts,["a","b","c"],"double");

[name,type] = getvartype(opts);
fprintf("%s: %s\n", [name; type])
```

```
a: double
b: double
c: double
```

Connect to the stream to export data to `numericTriangles`.

```
outKS = kafkaStream("kafka.host.com",9092,"numericTriangles", ...
    ExportOptions=opts)
```

```
outKS = 

  KafkaStream with properties:

                  Topic: "numericTriangles"
                  Group: "85c42e39-695d-467a-86f0-f0095792e7de"
                  Order: EventTime
                   Host: "kafka.host.com"
                   Port: 9092
      ConnectionTimeout: 30
         RequestTimeout: 61
          ImportOptions: "None"
          ExportOptions: "Source: string"
          PublishSchema: "true"
             WindowSize: 50
            KeyVariable: "key"
            KeyEncoding: "utf16"
                KeyType: "text"
           KeyByteOrder: "BigEndian"
           BodyEncoding: "utf8"
             BodyFormat: "JSON"
              ReadLimit: "Size"
    TimestampResolution: "Milliseconds"
```

Export the timetable to the new stream. The triangle side lengths in this stream are of type `double`.

```
writetimetable(outKS,tt);
```

Input Arguments

`opts` — Event stream export options
`ExportOptions` object

Event stream export options, specified as an `ExportOptions` object.

`selection` — Selected variables
character vector | string scalar | cell array of character vectors | string array

Selected variables, specified as a character vector, string scalar, cell array of character vectors, or string array.

Variable names must be a subset of the names recognized by the `opts` object.

Example: `'FanID'`

Example: `"FanID"`

Example: `{'FanID','vMotor'}`

Example: `["FanID" "vMotor"]`

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

Output Arguments

`type` — Data types of variables
string array

Data types of variables exported to the stream, returned as a string array.

Each element of `type` specifies the data type of a variable in the stream. If you specified `selection`, the order of the returned types matches the order of the variables named in `selection`. Otherwise, the order matches the order of the variable names in the timetable row used to create `opts` in the call to `detectExportOptions`.

`name` — Names of variables
string array

Names of variables exported to the stream, returned as a string array.

Each element of `name` specifies the name of a variable in the stream. The number and order of the returned names matches the number and order of the data types returned by `type`.

Version History

Introduced in R2022b

See Also

`setvartype` | `detectExportOptions`

identifyingName

Package: `matlab.io.stream.event`

Event stream name

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
name = identifyingName(stream)
```

Description

`name = identifyingName(stream)` returns the name that uniquely identifies an event stream. If the stream provider is Kafka, this name is the name of the topic that the stream is connected to.

Examples

Get Name of Kafka Topic

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Create an object for reading from and writing to the Kafka topic `CoolingFan`.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan")
```

```
ks = 

  KafkaStream with properties:

                Topic: "CoolingFan"
              GroupID: "425b8750-87ee-4d86-8e06-6f1a1ebfe009"
                Order: EventTime
                 Host: "kafka.host.com"
                 Port: 9092
    ConnectionTimeout: 30
       RequestTimeout: 61
    NumTimeoutRetries: 1
                State: Idle
           WindowSize: 0
          KeyVariable: "key"
          KeyEncoding: "utf16"
              KeyType: "text"
         KeyByteOrder: "BigEndian"
         BodyEncoding: "utf8"
           BodyFormat: "JSON"
           ImportBody: 1
        ImportOptions: [0×0 matlab.io.stream.event.ImportOptions]
        ExportOptions: [1×1 matlab.io.stream.event.ExportOptions]
```

```
          PublishSchema: 1
              ReadLimit: "Size"
    TimestampResolution: "Milliseconds"
             WindowUnit: None
                   Name: "CoolingFan"
```

Get the name of the Kafka topic.

```
identifyingName(ks)
```

```
ans = 

    "CoolingFan"
```

Input Arguments

`stream` — Object connected to event stream
`KafkaStream` object | `InMemoryStream` object | `TestStream` object

Object connected to an event stream, specified as a `KafkaStream`, `InMemoryStream`, or `TestStream` object.

Version History

Introduced in R2022b

See Also

`detectImportOptions`

ImportOptions

Import options for event stream

Note This object requires Streaming Data Framework for MATLAB® Production Server™.

Description

An `ImportOptions` object specifies how MATLAB imports tabular data from event streams. The object contains properties that control the data import process, including handling of errors and missing data.

Creation

You can create an `ImportOptions` object by using either the `detectImportOptions` function or the `eventStreamImportOptions` function. The preferred way is to use `detectImportOptions`.

- Use `detectImportOptions` to detect and populate the import properties based on the contents of the event stream specified by `stream`.

  ```
  opts = detectImportOptions(stream)
  ```
- Use `eventStreamImportOptions` to create import properties by specifying import options as name-value arguments.

  ```
  opts = eventStreamImportOptions(Name1=Value1,...,NameN=ValueN)
  ```

Properties

`SelectedVariableNames` — Subset of variables to import
character vector | string scalar | cell array of character vectors | string array

Subset of variables to import, specified as a character vector, string scalar, cell array of character vectors, or string array.

`SelectedVariableNames` must be a subset of names contained in the `VariableNames` property. By default, `SelectedVariableNames` contains all the variable names from the `VariableNames` property, which means that all variables are imported.

Use the `SelectedVariableNames` property to import only the variables of interest. Specify a subset of variables using the `SelectedVariableNames` property and use the `readtimetable` function to import only that subset.

Example: `opts.SelectedVariableNames = "x"` imports only the variable `x` from the event stream when you use `readtimetable` to import event stream data into a timetable.

Data Types: `char` | `string`

`VariableNames` — Variable names
cell array of character vectors | string array

Variable names, specified as a cell array of character vectors or string array. The `VariableNames` property contains the names to use when importing variables from the event stream into a timetable.

These variable names must exist in the stream. If the modified variable name is not in the stream, the import operation fails.

Example: `io.VariableNames` returns the current variable names in the event stream.

Example: `io.VariableNames(3) = {'Mass'}` changes the name of the third variable to `Mass`.

Data Types: `cell` | `string`

`VariableTypes` — Data type of variables
cell array of character vectors | string array

Data type of variables, specified as a cell array of character vectors or string array containing a set of valid data type names. The `VariableTypes` property designates the data types to use when importing variables from the event stream into a timetable.

The import operation attempts to convert the values in the stream to these types. This operation succeeds only when the conversion between primitive types is known and unambiguous, such as:

- Integer to string conversions
- Conversions where the constructor of the target type can accept a variable of that type in the stream

To update the `VariableTypes` property, use the `setvartype` function.

Example: `io.VariableTypes` returns the data types that stream variables have after they are imported into MATLAB timetables, which are the types of the corresponding timetable columns.

Example: `io = setvartype(io,"vMotor","int32")` changes the data type of the `vMotor` variable to `int32`.

Data Types: `cell` | `string`

`KeyVariable` — Key variable name
`key` (default) | string scalar | character vector

Name of the key variable in the event stream, specified as a string scalar or character vector. The default value is `key`.

Data Types: `string` | `char`

Object Functions

setvartype	Set data types used to import and export variables to stream

Examples

Create Import Options from Event Stream Data

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `Triangles` with JSON-encoded event value `'{"triangle":{"x":"3","y":"4","z":"5"}}'`. Each event value contains a variable `"triangle"`, which is a structure of side lengths `"x"`, `"y"`, and `"z"`. The side lengths are integers but are encoded as strings.

`readtimetable` creates a `triangle` table column for this variable and sets the column value to this structure:

```
triangle = 

  struct with fields:

    x: "3"
    y: "4"
    z: "5"
```

`ImportOptions` enables you to change the types of the values in the imported structure and select which fields to import.

Create a `KafkaStream` object connected to the `Triangles` topic.

```
ks = kafkaStream("kafka.host.com",9092,"Triangles");
```

Create an `ImportOptions` object from the Kafka stream object. The data type of the length of each side is `string`.

```
opts = detectImportOptions(ks)
```

```
opts = 

  ImportOptions with properties:

            VariableNames: ["triangle/x"    "triangle/y"    "triangle/z"]
            VariableTypes: ["string"    "string"    "string"]
              KeyVariable: "key"
    SelectedVariableNames: ["triangle/x"    "triangle/y"    "triangle/z"]
```

To perform mathematical operations on the imported data, update the data type of variables to `double`. Because the side length variables are nested within `"triangle"`, use a forward slash (`"/"`) to specify the path to these variables.

```
opts = setvartype(opts, ["triangle/x", "triangle/y", "triangle/z"], "double")
```

```
opts = 

  ImportOptions with properties:

            VariableNames: ["triangle/x"    "triangle/y"    "triangle/z"]
            VariableTypes: ["double"    "double"    "double"]
              KeyVariable: "key"
    SelectedVariableNames: ["triangle/x"    "triangle/y"    "triangle/z"]
```

Update the `ImportOptions` property of the `KafkaStream` object.

```
ks.ImportOptions = opts
```

```
ks = 

  KafkaStream with properties:

                Topic: "Triangles"
                Group: "85c42e39-695d-467a-86f0-f0095792e7de"
                Order: EventTime
                 Host: "kafka.host.com"
                 Port: 9092
    ConnectionTimeout: 30
       RequestTimeout: 61
```

```
          ImportOptions: "Import to MATLAB types"
          ExportOptions: "Source: function eventSchema"
          PublishSchema: "true"
             WindowSize: 50
            KeyVariable: "key"
            KeyEncoding: "utf16"
                KeyType: "text"
           KeyByteOrder: "BigEndian"
           BodyEncoding: "utf8"
             BodyFormat: "JSON"
              ReadLimit: "Size"
    TimestampResolution: "Milliseconds"
```

When importing the triangles, `readtimetable` converts the side lengths to `double` values.

```
tt = readtimetable(ks);
tt(1,:).triangle
```

```
ans = 

  struct with fields:

    x: 3
    y: 4
    z: 5
```

Create Import Options by Specifying Import Variable Names and Types

Create a schema for importing data from an event stream into MATLAB by specifying variable names and their data types to use during the import.

```
names = ["x","symbol"];
types = ["double","string"];
```

Construct an `ImportOptions` object using this data import schema.

```
opts = eventStreamImportOptions(VariableNames=names,VariableTypes=types)
```

```
opts = 

  ImportOptions with properties:

            VariableNames: ["x"    "symbol"]
            VariableTypes: ["double"    "string"]
              KeyVariable: [0×0 string]
    SelectedVariableNames: ["x"    "symbol"]
```

Apply the import options when creating a `KafkaStream` object.

```
ks = kafkaStream("kafka.host.com",9092,"Your_Kafka_Topic",ImportOptions=opts);
```

Import the data. The `"Your_Kafka_Topic"` topic must have events with exactly two variables, `x` and `symbol`. In addition, the types of these variables must be convertible to double and string, respectively. Otherwise, `readtimetable` throws an error.

```
tt = readtimetable(ks);
```

Version History

Introduced in R2022b

See Also

`detectImportOptions` | `setvartype` | `readtimetable`

inMemoryStream

Create connection to event stream hosted by MATLAB without schema processing applied

Note This object requires Streaming Data Framework for MATLAB® Production Server™.

Description

The `inMemoryStream` function creates an `InMemoryStream` object, which you can use to test reading from and writing to event streams hosted by MATLAB. Unlike the `TestStream` object, `InMemoryStream` objects do not apply schema processing when reading and writing timetable data. The data in an `inMemoryStream` object disappears when you exit MATLAB.

Creation

Syntax

```
stream = inMemoryStream
stream = inMemoryStream(Rows=numevents)

stream = inMemoryStream(Duration=timespan)

stream = inMemoryStream( ___ ,Name=Value)
```

Description

Row-Based Event Window

`stream = inMemoryStream` creates a default `InMemoryStream` object connected to an event stream hosted by MATLAB that does not apply schema processing. The object reads 50 stream event rows at a time.

`stream = inMemoryStream(Rows=numevents)` creates an `InMemoryStream` object that reads `numevents` stream event rows at a time.

Duration-Based Event Window

`stream = inMemoryStream(Duration=timespan)` creates an `InMemoryStream` object that reads stream events occurring during the specified timestamp span, `timespan`.

Additional Options

`stream = inMemoryStream( ___ ,Name=Value)` sets event stream properties on page 10-46 using one or more name-value arguments and any of the previous syntaxes.

Input Arguments

`numevents` — Number of events in event window
50 (default) | positive integer

Number of events in the event window, specified as a positive integer. `Rows=numevents` specifies the number of rows that a call to the `readtimetable` function returns. If there are less than the number of specified rows available for reading, then `readtimetable` times out and returns an empty timetable.

Example: `Rows=500` specifies that each call to `readtimetable` returns a timetable with 500 rows.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`timespan` — Timestamp span in event window
0 (default) | duration scalar

Timestamp span in the event window, specified as a duration scalar. `Duration=timespan` determines the events that the `readtimetable` function returns based on their timestamp. `timespan` specifies the difference between the last and first timestamps of events in the event window.

Example: `Duration=minutes(1)` specifies that each call to `readtimetable` returns a timetable that has one minute's worth of events, where the timestamp of the last event is no more than one minute later than the timestamp of the first event.

Data Types: `duration`

Properties

`Name` — Event stream name
string scalar

Event stream name, specified as a string scalar or character vector.

You cannot set the value of this property or use it as an input argument during object creation.

Example: `5cb30967-46fd-4058-8be7-704e4dbccc8d`

Data Types: `string`

`WindowSize` — Event window size
50 (default) | duration scalar | positive integer

This property is read-only.

Event window size, specified as a fixed amount of time (using the `timespan` argument) or a fixed number of messages (using the `numevents` argument).

Data Types: `duration` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ReadLimit` — Wait strategy
`"Size"` (default) | `"Time"`

Strategy to wait for a response from the stream, specified as one of these values:

- `"Size"` — Client prioritizes filling the event window. Using this strategy, the client might wait longer than the `RequestTimeout` time period as long as it is still receiving the expected number of messages. The default number of messages is 50. If the client receives no messages within the `RequestTimeout` time period, it no longer waits.

- `"Time"` — Client strictly adheres to the `RequestTimeout` limit, even if it has not received the expected number of messages. `RequestTimeout` specifies the amount of time the stream object waits between receiving events. If the stream is actively receiving data, it does not time out during that operation.

Note This object does not implement the `ReadLimit` property and does not have a `RequestTimeout` property. It is provided for compatibility with other stream connector objects. By setting the wait strategy in this object, you can more easily update your code to switch between objects that do implement this property, such as `KafkaStream`.

`TimestampResolution` — Unit of event timestamp
`"Milliseconds"` (default) | `"Seconds"` | `"Minutes"` | `"Hours"` | `"Days"`

Unit of event timestamp, specified as one of these values:

- `"Milliseconds"`
- `"Seconds"`
- `"Minutes"`
- `"Hours"`
- `"Days"`

Interpret the event timestamp as the number of corresponding units before or after the Unix epoch.

Data Types: `string` | `char`

Event Key and Body Encoding

`KeyVariable` — Name of key variable
`key` (default) | string scalar | character vector

Name of the key variable in the event stream, specified as a string scalar or character vector.

Data Types: `string` | `char`

KeyEncoding — Character encoding format for bits in event key
`utf8` (default) | `utf16` | `base64` | `uint8`

Character encoding format used to interpret the bits in an event key, specified as one of the following:

- `utf8` — UTF-8 encoding format
- `utf16` — UTF-16 encoding format
- `base64`— Base 64 encoding format
- `uint8` — Eight-bit unsigned binary bytes

If `KeyEncoding` is `utf8` or `utf16`, then the `KeyType` property must be `text`. If `KeyEncoding` is `base64` or `uint8`, then `KeyType` must be one of the numeric encoding formats.

KeyType — Character encoding scheme for bytes in event key
`text` (default) | `utf16` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `int64` | `uint64` | `single` | `double`

Character encoding scheme used to interpret the bytes in an event key, specified as one of these values:

- `uint8` — One-byte unsigned integer
- `int8` — One-byte signed integer
- `uint16` — Two-byte unsigned integer
- `int16` — Two-byte signed integer
- `uint32` — Four-byte unsigned integer
- `int32` — Four-byte signed integer
- `uint64` — Eight-byte unsigned integer
- `int64` — Eight-byte signed integer
- `single` — Single-precision IEEE 754 floating point number
- `double` — Double-precision IEEE 754 floating point number
- `text` — String

If `KeyType` is `text`, then the `KeyEncoding` property must be either `utf8` or `utf16`. If `KeyType` is any of the other numeric encoding formats, then `KeyEncoding` must be either `base64` or `uint8`.

`KeyByteOrder` — Order for storing bits in event key
`BigEndian` (default) | `LittleEndian` | `MatchHost` | `NotApplicable`

Order for storing bits in the event key, specified as one of the following.

- `LittleEndian` — Least significant bit is stored first
- `BigEndian` — Most significant bit is stored first
- `MatchHost`— Bits are stored in the same order as is used by the host computer on which the streaming data framework is running
- `NotApplicable` — Not an integer key

This property is applicable only for integer keys and not applicable to floating point or text keys.

`BodyEncoding` — Character encoding format for bits in event body
`utf8` (default) | `uint8` | `utf16` | `base64`

Character encoding format used to interpret the bits in the event body, specified as one of the following:

- `utf8` — UTF-8 encoding format
- `utf16` — UTF-16 encoding format
- `base64`— Base 64 encoding format
- `uint8` — Eight-bit unsigned binary bytes

This property determines the size and encoding of the bytes used in the event body, which are in the format specified by `BodyFormat`.

`BodyFormat` — Format of bytes in event body
`JSON` (default) | `Array` | `Text` | `Binary`

Format of bytes in event body, specified as one of the following:

- `JSON` — JSON string
- `Array` — MATLAB array
- `Text` — String data
- `Binary` — Binary data

Depending on the encoding specified by `BodyEncoding`, bytes can be larger than eight bits.

Object Functions

readtimetable	Read timetable from event stream
writetimetable	Write timetable to event stream
seek	Set read position in event stream
preview	Preview subset of events from event stream
identifyingName	Event stream name
detectImportOptions	Create import options based on event stream content
detectExportOptions	Create export options based on event stream content

Examples

Write and Preview Events for In-Memory Stream

Create an `InMemoryStream` object to preview events from and write events to an event stream hosted by MATLAB.

```
is = inMemoryStream
```

```
is = 

  InMemoryStream with properties:

                 Name: "7803fad7-64b3-4439-af40-1b0e139fd68a"
           WindowSize: 50
          KeyVariable: "key"
          KeyEncoding: "utf8"
              KeyType: "text"
         KeyByteOrder: "BigEndian"
         BodyEncoding: "uint8"
           BodyFormat: "Array"
            ReadLimit: "Size"
  TimestampResolution: "Milliseconds"
```

Write timetable data to the event stream.

```
load indoors
writetimetable(is,indoors)
```

Preview data from the timetable. Unlike `TestStream` and `KafkaStream`, objects the `InMemoryStream` object does not include a key column to identify the event source, because the data kept in memory disappears when you exit MATLAB.

```
preview(is)
```

```
ans =
  8×2 timetable

           Time           Humidity    AirQuality
    ___________________    ________    __________

    2015-11-15 00:00:24       36            80
    2015-11-15 01:13:35       36            80
    2015-11-15 02:26:47       37            79
    2015-11-15 03:39:59       37            82
    2015-11-15 04:53:11       36            80
    2015-11-15 06:06:23       36            80
    2015-11-15 07:19:35       36            80
    2015-11-15 08:32:47       37            80
```

Version History

Introduced in R2022b

See Also

`kafkaStream` | `testStream`

Topics

"Streaming Data Framework for MATLAB Production Server Basics" on page 11-2

isProperty

Package: `matlab.io.stream.event`

Determine if Kafka stream provider property is set

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
tf = isProperty(ks,name)
tf = isProperty(ks,name,cat)
[tf,prop] = isProperty( ___ )
[tf,prop,type] = isProperty( ___ )
```

Description

`tf = isProperty(ks,name)` returns a logical 1 (true) if the Kafka stream provider property `name` is set in the stream connector object `ks`. Otherwise, it returns logical 0 (false). The `getProviderProperties` function returns nonempty property data only for properties for which `isProperty` returns true.

You can specify multiple property names in `name`. The length of `tf` equals the number of string property names in `name`.

`tf = isProperty(ks,name,cat)` restricts the search for a property to within Kafka stream provider category `cat`.

`[tf,prop] = isProperty( ___ )` also returns the property names and categories, `prop`. Because properties can belong to multiple categories or be unset, `prop` might not be the same size as `tf`. You can return `prop` using any of the preceding syntaxes.

`[tf,prop,type] = isProperty( ___ )` also returns the data type of the property values, `type`. The `prop` and `type` arguments are always the same size.

Examples

Determine If Specific Kafka Stream Provider Properties Are Set

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Create a `KafkaStream` object connected to the Kafka host and also specify Kafka provider properties during object creation.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan", ...
"security.protocol","SSL","ssl.truststore.type","PEM", ...
"ssl.truststore.location","kafka-boston.pem","retention.ms",500);
```

Check if the `security.protocol` and `retention.ms` properties are set.

```
isProperty(ks,["security.protocol","retention.ms"])
```

```
ans =

  1×2 logical array

   1   1
```

Determine If Kafka Stream Provider Properties from Specific Category Are Set

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Create a `KafkaStream` object connected to the Kafka host and also specify Kafka provider properties during object creation.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan", ...
"security.protocol","SSL","ssl.truststore.type","PEM", ...
"ssl.truststore.location","kafka-boston.pem","retention.ms",500);
```

Check if the `security.protocol` and `retention.ms` properties belong to the `CreateTopic` category.

```
isProperty(ks,["security.protocol" "retention.ms"],"CreateTopic")
```

```
ans =

  1×2 logical array

   0   1
```

Return Data for Set Kafka Stream Provider Properties

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Create a `KafkaStream` object connected to the Kafka host and also specify Kafka provider properties during object creation.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan", ...
"security.protocol","SSL","ssl.truststore.type","PEM", ...
"ssl.truststore.location","kafka-boston.pem","retention.ms",500);
```

Check if the `security.protocol` and `retention.ms` properties belong to the `CreateTopic` category and return the property names, categories, and data types. Because `security.protocol` is set in two different categories, it appears twice in the `prop` and `type` outputs.

```
[tf,prop,type] = isProperty(ks,["security.protocol" "retention.ms"]);
tf
name = {prop.name}
cat = {prop.category}
type
```

```
tf =

  1×2 logical array

   1   1


name =

  1×3 cell array

    {["security.protocol"]}    {["security.protocol"]}    {["retention.ms"]}


cat =

  1×3 cell array

    {["Consumer"]}    {["Producer"]}    {["CreateTopic"]}


type =

  1×3 string array

    "string"    "string"    "string"
```

Input Arguments

ks — Object connected to Kafka stream topic
`KafkaStream` object

Object connected to a Kafka stream topic, specified as a `KafkaStream` object.

name — Kafka stream provider property names
string scalar | character vector | string array | cell array of character vectors

Kafka stream provider property names, specified as a string scalar, character vector, string array, or cell array of character vectors. If a property is not set in `ks`, then `isProperty` returns a logical `0` (false) for that property in `tf`, and the optional `prop` and `type` arguments do not return data for this property.

Example: `tf = isProperty(ks,"retention.ms")` returns whether the `retention.ms` property is set in `ks`.

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

cat — Kafka stream provider category names
string scalar | character vector | string array | cell array of character vectors

Kafka stream provider category names, specified as a string scalar, character vector, string array, or cell array of character vectors.

The specified categories must be present in `ks`. To get a list of valid categories, use the `ks.PropertyCategories` property.

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

Output Arguments

`tf` — Provider property set indicator
logical vector

Provider property set indicator, returned as a logical array of these values:

- `1` — The property in the corresponding position of `name` is set in `ks`.
- `0` — The property in the corresponding position of `name` is not set in `ks`.

`prop` — Kafka stream provider property names and categories
structure array

Kafka stream provider property names and categories, returned as a structure array. Each structure corresponds to a provider property in `ks` and has these fields:

- `name` — Provider property name, returned as a string
- `category` — Category that the provider property belongs to, returned as a string

Because properties can belong to more than one category, the category and name uniquely identity a property.

`type` — Data types of Kafka stream provider property values
string scalar | string array

Data types of Kafka stream provider property values, returned as a string scalar or string array. Each string in `type` specifies the data type of a set property returned in the corresponding position of `prop`.

Version History

Introduced in R2022b

See Also

`getProviderProperties` | `setProviderProperties` | `categoryList`

Topics
"Connect to Secure Kafka Cluster" on page 11-9

kafkaStream

Create connection to event stream in Kafka topic

Note This object requires Streaming Data Framework for MATLAB® Production Server™.

Description

The `kafkaStream` function creates a `KafkaStream` object that connects to a Kafka topic and reads and writes event streams from that topic.

An event consists of three parts:

- Key — Identifies event source
- Timestamp — Indicates time at which event occurred
- Body — Contains event data specified as an unordered set of (name, value) pairs

After creating a `KafkaStream` object, use the `readtimetable` function to read the events into a timetable or the `writetimetable` function to write a timetable to the stream.

`readtimetable` converts events into rows of a timetable. The names in the event body become the timetable column names, the value associated with each name becomes the column value in the event row, and the event timestamp becomes the row timestamp. `writetimetable` converts rows of a timetable into events in a stream.

Creation

Syntax

```
ks = kafkaStream(host,port,topic)
ks = kafkaStream(host,port,topic,Rows=numevents)

ks = kafkaStream(host,port,topic,Duration=timespan)

ks = kafkaStream(___,propname1,propval1,...,propnameN,propvalN)
ks = kafkaStream(___,Name=Value)
```

Description

Row-Based Event Window

`ks = kafkaStream(host,port,topic)` creates a default `KafkaStream` object connected to a Kafka topic at the specified hostname and port. This syntax sets the `Host`, `Port`, and `Topic` properties to `host`, `port`, and `topic`, respectively. The object reads 50 stream event rows at a time.

`ks = kafkaStream(host,port,topic,Rows=numevents)` creates a `KafkaStream` object that reads `numevents` stream event rows at a time.

Duration-Based Event Window

`ks = kafkaStream(host,port,topic,Duration=timespan)` creates a `KafkaStream` object that reads stream events occurring during the specified timestamp span `timespan`.

Additional Options

`ks = kafkaStream( ___ ,propname1,propval1,...,propnameN,propvalN)` sets Kafka provider properties using any of the previous syntaxes.

`ks = kafkaStream( ___ ,Name=Value)` specifies event stream options using one or more name-value arguments on page 10-57. You can also set properties on page 10-58 using name-value arguments. You can use these name-value arguments and properties to specify how events are converted to and from timetables.

Input Arguments

`numevents` — Number of events in event window

50 (default) | positive integer

Number of events in the event window, specified as a positive integer. `Rows=numevents` specifies the number of rows that a call to the `readtimetable` function returns. If less than the number of specified rows are available for reading, then `readtimetable` times out and returns an empty timetable.

`readtimetable` does not return until it processes all events in the window, so windows with large row values can block other processes from continuing. To configure a timeout period to prevent blocking, use the `ReadLimit` property.

Example: `Rows=500` specifies that each call to `readtimetable` returns a timetable with 500 rows.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`timespan` — Timestamp span in event window

0 (default) | duration scalar

Timestamp span in the event window, specified as a duration scalar. `Duration=timespan` determines the events that the `readtimetable` function returns based on their timestamp. `timespan` specifies the difference between the last and first timestamps of events in the event window.

`readtimetable` does not return until it processes all events in the window, so windows with large durations can block other processes from continuing. To configure a timeout period to prevent blocking, use the `ReadLimit` property.

Example: `Duration=minutes(1)` specifies that each call to `readtimetable` returns a timetable that has one minute's worth of events, where the timestamp of the last event is no more than one minute later than the timestamp of the first event.

Data Types: `duration`

`propname` — Name of Kafka provider property

character vector | string scalar

Name of a Kafka provider property, specified as a character vector or string scalar. Use single or double quotes around `propname`. Kafka property names always contain at least one dot character, for example, `retention.ms`. For a list of Kafka properties, see the Kafka documentation: https://kafka.apache.org/documentation/#configuration.

The value of the property, `propval`, must follow the property name. Specify the property name and its corresponding value as a comma-separated pair.

Example: `kafkaStream(host,port,topic,"security.protocol","SASL_SSL")` sets the Kafka configuration property `security.protocol` to `SASL_SSL`.

`propval` — Value of Kafka provider property
any supported MATLAB data type

Value of a Kafka provider property. For a list of Kafka properties and their values, see the Kafka documentation: https://kafka.apache.org/documentation/#configuration.

The value of the property must follow the property name `propname`. Specify the property name and its corresponding value as a comma-separated pair. You can specify `propval` as any supported MATLAB data type, but it must be possible to convert that value to a string.

Example: `kafkaStream(host,port,topic,"sasl.mechanism","SCRAM-SHA-512")` sets the value of the Kafka configuration property `sasl.mechanism` to `SCRAM-SHA-512`.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Grace Period

`GracePeriod` — Length of time to wait for messages
`0` (default) | real scalar | duration scalar

Length of time, in `GraceUnit` units, to wait for messages in the requested event window, specified as a real scalar or duration scalar. The `KafkaStream` object waits until the end of the grace period to return events read from the stream. The `GracePeriod` and `GraceUnit` arguments together set the `GracePeriod` property.

This argument applies only for objects with duration-based event windows, that is, `KafkaStream` objects created using the `timespan` argument. For objects created using the `numevents` argument, the grace period is ignored.

Example: `10`

Example: `minutes(10)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`GraceUnit` — Unit of time for grace period
`"Seconds"` (default) | `"Milliseconds"` | `"Minutes"` | `"Hours"` | `"Days"`

Unit of time for the grace period specified by the `GracePeriod` name-value argument, specified as one of these values:

- `"Milliseconds"`

- "Seconds"
- "Minutes"
- "Hours"
- "Days"

The GracePeriod and GraceUnit arguments together set the GracePeriod property.

The KafkaStream object converts GracePeriod duration scalars to the units specified by GraceUnit. For example, suppose you specify a two-minute grace period using the minutes function but set the units to seconds. The GracePeriod property displays the grace period in seconds.

```
ks = kafkaStream(host,port,topic,Duration=minutes(10), ...
                 GracePeriod=minutes(2),GraceUnit="Seconds")
```

```
ks = 

  KafkaStream with properties:
            ...
            GracePeriod: "120 Seconds"
            ...
```

This argument applies only for objects with duration-based event windows, that is, KafkaStream objects created using the timespan argument. For objects created using the numevents argument, the grace period is ignored.

Data Types: string | char

Schema

ImportSchema — Rules for converting event data to MATLAB data types
JSON string in event schema format

Rules for converting event data to MATLAB data types, specified as a JSON string in event schema format. You can specify an event schema more easily using the ImportOptions property.

ExportSchema — Rules for converting MATLAB data types to event data
JSON string in event schema format

Rules for converting MATLAB data types to event data, specified as a JSON string in event schema format. You can specify an event schema more easily using the ExportOptions property.

Properties

Host — Hostname of Kafka server
character vector | string scalar

Hostname of the Kafka server, specified as a character vector or string scalar.

Example: '144.213.5.7' or 'localhost'

Data Types: char | string

Port — Port number of Kafka server
integer in range [0, 65,535]

Port number of the Kafka server, specified as an integer in the range [0, 65,535].

Example: 9092

Topic — Kafka topic name
character vector | string scalar

Kafka topic name, specified as a character vector or string scalar.

Example: "CoolingFan"

Data Types: char | string

Group — Kafka consumer group ID
UUID (default) | character vector | string scalar

Kafka consumer group ID, specified as a character vector or string scalar.

Multiple Kafka consumers can belong to the same consumer group. In that case, Kafka shares data between the consumers in the group so that no two consumers in the same group ever receive the same messages. By default, every `kafkaStream` object has a unique consumer group ID, which allows multiple consumers to read from the same topic independently.

Data Types: char | string

Order — Event order
"EventTime" (default) | "IngestTime"

Strategy to order events in the stream, specified as one of these values:

- `"EventTime"` — Order events based on the time that they occur. Ensures event-time chronological order even when events arrive out of order at the Kafka server.
- `"IngestTime"` — Order events based on the time that they appear in the stream.

You cannot set the value of this property after object creation.

Data Types: string | char

GracePeriod — Time to wait for messages
"0 Seconds" (default) | string scalar

This property is read-only.

Time that the `KakfaStream` object waits for messages, specified as a string scalar of the form `"Length Units"`, where:

- *`Length`* is the length of the grace period, as specified by the `GracePeriod` argument during object creation.
- *`Units`* is the units of the grace period, as specified by the `GraceUnits` argument during object creation.

When you create the object, if you do not specify a grace period, then the `GracePeriod` property is set to `"0 Seconds"` (no grace period).

Example: "10 Minutes"

Data Types: string

WindowSize — Event window size
50 (default) | duration scalar | positive integer

This property is read-only.

Event window size, specified as a fixed amount of time (using the `timespan` argument) or a fixed number of messages (using the `numevents` argument).

Data Types: `duration` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ReadLimit` — Wait strategy
`"Size"` (default) | `"Time"`

Strategy to wait for a response from the stream, specified as one of these values:

- `"Size"` — Client prioritizes filling the event window. Using this strategy, the client might wait longer than the `RequestTimeout` time period as long as it is still receiving the expected number of messages. The default number of messages is 50. If the client receives no messages within the `RequestTimeout` time period, it no longer waits.
- `"Time"` — Client strictly adheres to the `RequestTimeout` limit, even if it has not received the expected number of messages. `RequestTimeout` specifies the amount of time the stream object waits between receiving events. If the stream is actively receiving data, it does not time out during that operation.

`TimestampResolution` — Unit of event timestamp
`"Milliseconds"` (default) | `"Seconds"` | `"Minutes"` | `"Hours"` | `"Days"`

Unit of event timestamp, specified as one of these values:

- `"Milliseconds"`
- `"Seconds"`
- `"Minutes"`
- `"Hours"`
- `"Days"`

Interpret the event timestamp as the number of corresponding units before or after the Unix epoch.

Data Types: `string` | `char`

Connection and Request Timeouts

`ConnectionTimeout` — Number of seconds to wait for initial response from Kafka host
30 (default) | positive integer

Number of seconds that a client waits for the initial response from the Kafka host, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`RequestTimeout` — Number of seconds to wait before terminating request
61 (default) | positive integer

Number of seconds to wait before terminating a request, specified as a positive integer. The wait time includes connecting to the Kafka host as well as data transfer between the Kafka host and the client.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Import and Export Options

`ImportOptions` — Rules for transforming stream events into MATLAB data
`ImportOptions` object

Rules for transforming stream events into MATLAB data, specified as an `ImportOptions` object. This object controls the import of stream events into MATLAB.

`ExportOptions` — Rules for transforming MATLAB data into stream events
`ExportOptions` object

Rules for transforming MATLAB data into stream events, specified as an `ExportOptions` object. This object controls the export of MATLAB data into streams.

`PublishSchema` — Flag to indicate whether export schema is written to output stream
`true` (default) | `false`

Flag to indicate whether the export schema is written to the output stream, specified as a logical scalar.

The schema is embedded in each event, which can significantly increase the size of the event. If downstream applications do not require the schema, set this flag to false to reduce the number of bytes in your stream.

Data Types: `logical`

Event Key and Body Encoding

`KeyVariable` — Name of key variable
`key` (default) | string scalar | character vector

Name of the key variable in the event stream, specified as a string scalar or character vector.

Data Types: `string` | `char`

`KeyEncoding` — Character encoding format for bits in event key
`utf16` (default) | `utf8` | `base64` | `uint8`

Character encoding format used to interpret the bits in an event key, specified as one of the following:

- `utf8` — UTF-8 encoding format
- `utf16` — UTF-16 encoding format
- `base64`— Base 64 encoding format
- `uint8` — Eight-bit unsigned binary bytes

If `KeyEncoding` is `utf8` or `utf16`, then the `KeyType` property must be `text`. If `KeyEncoding` is `base64` or `uint8`, then `KeyType` must be one of the numeric encoding formats.

`KeyType` — Character encoding scheme for bytes in event key
`utf16` (default) | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `int64` | `uint64` | `single` | `double` | `text`

Character encoding scheme used to interpret the bytes in an event key, specified as one of these values:

- `uint8` — One-byte unsigned integer
- `int8` — One-byte signed integer
- `uint16` — Two-byte unsigned integer
- `int16` — Two-byte signed integer
- `uint32` — Four-byte unsigned integer
- `int32` — Four-byte signed integer
- `uint64` — Eight-byte unsigned integer
- `int64` — Eight-byte signed integer
- `single` — Single-precision IEEE 754 floating point number
- `double` — Double-precision IEEE 754 floating point number
- `text` — String

If `KeyType` is `text`, then the `KeyEncoding` property must be either `utf8` or `utf16`. If `KeyType` is any of the other numeric encoding formats, then `KeyEncoding` must be either `base64` or `uint8`.

`KeyByteOrder` — Order for storing bits in event key
`BigEndian` (default) | `LittleEndian` | `MatchHost` | `NotApplicable`

Order for storing bits in the event key, specified as one of the following.

- `LittleEndian` — Least significant bit is stored first
- `BigEndian` — Most significant bit is stored first
- `MatchHost`— Bits are stored in the same order as is used by the host computer on which the streaming data framework is running
- `NotApplicable` — Not an integer key

This property is applicable only for integer keys and not applicable to floating point or text keys.

`BodyEncoding` — Character encoding format for bits in event body
`utf8` (default) | `uint8` | `utf16` | `base64`

Character encoding format used to interpret the bits in the event body, specified as one of the following:

- `utf8` — UTF-8 encoding format
- `utf16` — UTF-16 encoding format
- `base64`— Base 64 encoding format
- `uint8` — Eight-bit unsigned binary bytes

This property determines the size and encoding of the bytes used in the event body, which are in the format specified by `BodyFormat`.

`BodyFormat` — Format of bytes in event body
`JSON` (default) | `Array` | `Text` | `Binary`

Format of bytes in event body, specified as one of the following:

- `JSON` — JSON string

- `Array` — MATLAB array
- `Text` — String data
- `Binary` — Binary data

Depending on the encoding specified by `BodyEncoding`, bytes can be larger than eight bits.

Object Functions

Import and Export

readtimetable	Read timetable from event stream
writetimetable	Write timetable to event stream
seek	Set read position in event stream
preview	Preview subset of events from event stream
identifyingName	Event stream name
detectImportOptions	Create import options based on event stream content
detectExportOptions	Create export options based on event stream content

Kafka Stream Operations

readevents	Read raw events from Kafka stream without schema processing applied
flush	Reset read window boundaries
stop	Stop processing event streams from Kafka topic
loggederror	Error information for Kafka stream operation
createTopic	Create topic in Kafka cluster
deleteTopic	Remove topic from Kafka cluster

Kafka Provider Properties

categoryList	Kafka stream provider property list
getProviderProperties	Kafka stream configuration property data
setProviderProperties	Set properties specific to Kafka configuration
isProperty	Determine if Kafka stream provider property is set

Examples

Set Kafka Security Protocol

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Assume that the Kafka host is configured to use SSL. To configure SSL communication between the Kafka host and the client, provide SSL configuration settings when creating an object for reading and writing to the Kafka topic.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan", ...
                "security.protocol","SASL_SSL", ...
                "ssl.truststore.type","PEM", ...
                "ssl.truststore.location","prodserver.pem")
```

```
ks =
  KafkaStream with properties:

                Topic: "CoolingFan"
                Group: "da576775-49c9-4de3-9955-2bdd9f963aa0"
                Order: EventTime
                 Host: "kafka.host.com"
                 Port: 9092
    ConnectionTimeout: 30
       RequestTimeout: 61
        ImportOptions: "Import to MATLAB types"
        ExportOptions: "Source: function eventSchema"
        PublishSchema: "true"
           WindowSize: 50
          KeyVariable: "key"
          KeyEncoding: "utf16"
              KeyType: "text"
         KeyByteOrder: "BigEndian"
         BodyEncoding: "utf8"
           BodyFormat: "JSON"
            ReadLimit: "Size"
  TimestampResolution: "Milliseconds"
```

Confirm which properties are set.

```
props = getProviderProperties(ks);
unique({props.name}')
```

```
ans =

  7×1 cell array

    {'auto.offset.reset'      }
    {'retention.ms'           }
    {'sasl.jaas.config'       }
    {'sasl.username'          }
    {'security.protocol'      }
    {'ssl.truststore.location'}
    {'ssl.truststore.type'    }
```

Read Specific Number of Kafka Messages

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Create an object connected to the `CoolingFan` topic and request only 10 messages instead of the default.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan",Rows=10)
```

```
ks =

  KafkaStream with properties:

                Topic: "CoolingFan"
                Group: "da576775-49c9-4de3-9955-2bdd9f963aa0"
                Order: EventTime
```

```
                 Host: "kafka.host.com"
                 Port: 9092
    ConnectionTimeout: 30
       RequestTimeout: 61
        ImportOptions: "Import to MATLAB types"
        ExportOptions: "Source: function eventSchema"
        PublishSchema: "true"
           WindowSize: 10
          KeyVariable: "key"
          KeyEncoding: "utf16"
              KeyType: "text"
         KeyByteOrder: "BigEndian"
         BodyEncoding: "utf8"
           BodyFormat: "JSON"
            ReadLimit: "Size"
  TimestampResolution: "Milliseconds"
```

Use the object to read 10 messages from the event stream into a timetable.

```
tt = readtimetable(ks)
```

```
tt =

  10×11 timetable

         timestamp          vMotor    wMotor    Tmass
    ____________________    ______    ______    ______

    31-Oct-2020 00:00:00    1.0909         0        25
    31-Oct-2020 00:00:00    1.1506     100.5     25.17
    31-Oct-2020 00:00:00    1.1739     190.9    25.223
    31-Oct-2020 00:00:00    1.1454    330.61     25.15
    31-Oct-2020 00:00:00    1.1346    382.77    25.122
    31-Oct-2020 00:00:00    1.1287    420.88    25.106
    31-Oct-2020 00:00:00    1.1253    454.55    25.096
    31-Oct-2020 00:00:00    1.1232     478.1     25.09
    31-Oct-2020 00:00:00    1.1217    500.16    25.086    ...
```

Version History

Introduced in R2022b

See Also

`testStream` | `inMemoryStream` | `readtimetable` | `preview` | `readevents` | `loggederror`

Topics

"Streaming Data Framework for MATLAB Production Server Basics" on page 11-2
"Process Kafka Events Using MATLAB" on page 11-5
"Connect to Secure Kafka Cluster" on page 11-9

External Websites

Kafka Introduction

loggederror

Package: `matlab.io.stream.event`

Error information for Kafka stream operation

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
txt = loggederror(ks)
```

Description

`txt = loggederror(ks)` return the text `txt` of the most recent error that is logged in the Kafka error log. To obtain all Kafka log information, see "Obtain Kafka Event Stream Log Files" on page 11-21.

Examples

Get Error Details for Kafka Stream Operation

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `RecamanSequence`.

Create a `KafkaStream` object connected to the `RecamanSequence` topic. If the host you specify when creating the `KafkaStream` object does not exist, an error occurs if you call `readtimetable`.

```
ks = kafkaStream("kafka.host123.com",9092,"RecamanSequence");
tt = readtimetable(ks);

loggederror(ks)
```

```
ans =

    '... - Failed to start connector with exception:
             ...
     '
```

Input Arguments

ks — Object connected to Kafka stream topic
`KafkaStream` object

Object connected to a Kafka stream topic, specified as a `KafkaStream` object.

Version History

Introduced in R2022b

See Also

`kafkaStream`

Topics

"Obtain Kafka Event Stream Log Files" on page 11-21

package

Package: `matlab.io.stream.event`

Package stream processing function into deployable archive configured by `EventStreamProcessor`

Note This function requires Streaming Data Framework for MATLAB® Production Server™ and MATLAB Compiler SDK™.

Syntax

```
filePath = package(esp)
filePath = package(esp,Name=Value)
```

Description

`filePath = package(esp)` uses the `eventStreamProcessor` object `esp` to create a MATLAB Compiler SDK project file and opens the **Production Server Compiler** app. Use this app to package the streaming function into a deployable archive for MATLAB Production Server.

`package` returns the full path to the project file.

`filePath = package(esp,Name=Value)` sets additional options for packaging the function.

For example, if you specify `OutputType="Archive"`, the `package` function returns a deployable archive (CTF file) instead of a project file.

Examples

Package Streaming Analytic Function for Deployment to MATLAB Production Server

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `Triangles`.

Also, assume that you have a streaming analytic function `classifyTriangle`.

Create a `KafkaStream` object connected to the `Triangles` topic.

```
ks = kafkaStream("kafka.host.com",9092,"Triangles");
```

Create an `EventStreamProcessor` object to run the `classifyTriangle` streaming analytic function.

```
esp = eventStreamProcessor(ks,@classifyTriangle);
```

You can use `EventStreamProcesor` functions such as `execute`, `startServer`, `start`, `stopServer`, and `stop` to iterate over and test the streaming analytic functions using a local test server. Then, you can package the streaming analytic function `classifyTriangle` into a deployable archive for deployment to MATLAB Production Server.

```
file = package(esp)
file = 

    "J:\classifyTriangle.prj"
```

The `package` function generates a MATLAB project file based on the `eventStreamProcessor` object, returns the path to this file, and opens this project file with the **Production Server Compiler** app. The project file contains values for:

- The streaming analytic function, `classifyTriangle.m`
- The entry point function, `streamfcn.m`
- The deployable archive, `classifyTriangle.ctf`

To modify the list of deployed functions or the name of the generated archive, see “Customize Application and Its Appearance” (MATLAB Compiler SDK).

In the **Production Server Compiler**, click **Package** to generate the deployable archive. You can deploy the generated archive to MATLAB Production Server. For more information on deploying to MATLAB Production Server, see “Deploy Archive to MATLAB Production Server”.

Input Arguments

esp — Object to process event streams
`EventStreamProcessor` object

Object to process event streams, specified as an `EventStreamProcessor` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `filePath = package(esp,OutputFolder='J:\')`

`OutputType` — File type
`Project` (default) | `Archive`

Type of file created by the package function, specified as one of these values.

- `Project` — Creates a `productionServerCompiler` project file and launches the **Production Server Compiler** app. You can create a CTF file using the **Production Server Compiler** app.
- `Archive` — Creates a CTF file.

Data Types: `char` | `string`

`StateStore` — Persistent storage connection name
string | character vector

Persistent storage connection name, specified as a string or character vector. You must specify a `StateStore` when using `InitialState` or when using a stateful stream function. The connection name must be known to the MATLAB Production Server instance to which the archive will be deployed. For more information on using a data cache for persistent storage, see “Data Caching Basics”.

Data Types: `char` | `string`

`ExtraFiles` — Additional files to include in archive
character vector | string scalar | string array

Additional files to include in the generated archive, specified as a character vector or string scalar for a single file or as a string array for multiple files.

Example: `ExtraFiles=["data.mat","/schema/registry/schema.json"]` includes the files `data.mat` and `schema.json` in the generated deployable archive.

Data Types: `char` | `string`

`OutputFolder` — Location of generated file
current folder (default) | string | character vector

Location of the generated file, specified as a string or character vector.

Example: `OutputFolder='J:\'` saves the generated file in `J:\`.

Data Types: `string` | `char`

`ArchiveName` — Name of generated deployable archive
`streamFcn` (default) | string | character vector

Name of the generated deployable archive, specified as a string or character vector.

Data Types: `char` | `string`

`OpenProject` — Flag to automatically open project in MATLAB
`true` (default) | `false`

Flag to automatically open the project in MATLAB, specified as logical `true` or `false`. This property is incompatible with `OutputType="Archive"`.

Data Types: `logical`

Version History
Introduced in R2022b

See Also
`streamingDataCompiler` | `eventStreamProcessor`

Topics
"Deploy Streaming Analytic Function to MATLAB Production Server" on page 11-17

preview

Package: `matlab.io.stream.event`

Preview subset of events from event stream

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
tt = preview(stream)
tt = preview(stream,ReadLimit="Time",RequestTimeout=rt)
```

Description

`tt = preview(stream)` returns a timetable containing the first eight events from an event stream without advancing the read position. Multiple calls to `preview` return the same set of events.

`tt = preview(stream,ReadLimit="Time",RequestTimeout=rt)` specifies the request timeout value `rt` when previewing data in an event stream. This syntax is valid only for `KafkaStream` objects.

Examples

Preview Event Data

Create an `inMemoryStream` object to read and write events to an event stream hosted by MATLAB.

```
is = inMemoryStream;
```

Write timetable data to the event stream.

```
load indoors
writetimetable(is,indoors)
```

Preview the first 8 events in the stream.

```
preview(is)
```

```
ans =

  8×2 timetable

           Time             Humidity    AirQuality
    ___________________    ________    __________

    2015-11-15 00:00:24       36           80
    2015-11-15 01:13:35       36           80
    2015-11-15 02:26:47       37           79
    2015-11-15 03:39:59       37           82
    2015-11-15 04:53:11       36           80
    2015-11-15 06:06:23       36            80
    2015-11-15 07:19:35       36            80
    2015-11-15 08:32:47       37            80
```

Set Timeout when Previewing Kafka Event Data

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `IndoorTemp`.

Create a `KafkaStream` object for reading from and writing to the `IndoorTemp` topic.

```
ks = kafkaStream("kafka.host.com",9092,"IndoorTemp");
```

Write timetable data to the event stream.

```
load indoors
writetimetable(ks,indoors)
```

Set a timeout value of 30 seconds when previewing the first 8 events.

```
preview(ks,ReadLimit="Time",RequestTimeout=30)
```

```
ans =

  8×2 timetable

            Time             Humidity    AirQuality
    ___________________    ________    __________

    2015-11-15 00:00:24       36            80
    2015-11-15 01:13:35       36            80
    2015-11-15 02:26:47       37            79
    2015-11-15 03:39:59       37            82
    2015-11-15 04:53:11       36            80
    2015-11-15 06:06:23       36            80
    2015-11-15 07:19:35       36            80
    2015-11-15 08:32:47       37            80
```

Input Arguments

`stream` — Object connected to event stream
`KafkaStream` object | `InMemoryStream` object | `TestStream` object

Object connected to an event stream, specified as a `KafkaStream`, `InMemoryStream`, or `TestStream` object.

`rt` — Number of seconds to wait before terminating request
61 (default) | positive integer

Number of seconds to wait before terminating a request that has not yet begun to transfer data, specified as a positive integer.

If the stream provider is Kafka, the wait time includes connecting to the Kafka host as well as data transfer between the Kafka host and the client.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History

Introduced in R2022b

See Also

`readtimetable` | `kafkaStream` | `inMemoryStream` | `testStream` | `seek`

readevents

Package: `matlab.io.stream.event`

Read raw events from Kafka stream without schema processing applied

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
event = readevents(ks)
```

Description

`event = readevents(ks)` returns a structure array that contains raw events from the Kafka stream `ks`. Each event in the stream creates an event structure in the resulting structure array. No schema processing is applied to the event.

Examples

Read Raw Events From Kafka

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `RecamanSequence`.

Create a `KafkaStream` object for reading from and writing to the `RecamanSequence` topic.

```
ks = kafkaStream("kafka.host.com",9092,"RecamanSequence")
```

```
ks = 

  KafkaStream with properties:

                  Topic: "RecamanSequence"
                  Group: "d89f5726-6abf-461d-a14e-4d40ab84c676"
                  Order: EventTime
                   Host: "kafka.host.com"
                   Port: 9092
      ConnectionTimeout: 30
         RequestTimeout: 61
          ImportOptions: "None"
          ExportOptions: "Source: function eventSchema"
          PublishSchema: "true"
             WindowSize: 50
            KeyVariable: "key"
            KeyEncoding: "utf16"
                KeyType: "text"
           KeyByteOrder: "BigEndian"
           BodyEncoding: "utf8"
             BodyFormat: "JSON"
```

```
              ReadLimit: "Size"
    TimestampResolution: "Milliseconds"
```

Read 50 events, which is the default number of events, from the `RecamanSequence` topic.

```
events = readevents(ks)
```

```
events = 

  50×1 struct array with fields:

    key
    value
    timestamp
```

`readevents` blocks other operations until it reads 50 messages or times out after 61 seconds of receiving no messages. To strictly limit blocking time to 61 seconds even if more are messages available, specify `ReadLimit=Time` in the call to `kafkaStream`. To change the timeout duration, for example, to 15 seconds, specify `RequestTimeout=15` in the call to the `KafkaStream` object, `ks`.

Input Arguments

ks — Object connected to Kafka stream topic
`KafkaStream` object

Object connected to a Kafka stream topic, specified as a `KafkaStream` object.

Output Arguments

event — Event information
structure array

Event information, returned as a structure array. Each structure in the array has these fields.

key — Event key
string array | positive integer

Event key as stored in Kafka, returned as a string array or integer. The key identifies the event source.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

value — Event value
byte array

Event value, specified as a byte array with a format and encoding determined by the `BodyFormat` and `BodyEncoding` properties of the stream object. The event value does not undergo schema processing and appears exactly as is stored in Kafka, for example, as a JSON string.

Data Types: `string` | `uint8` | `uint16`

timestamp — Event timestamp or ingest timestamp
datetime scalar

Timestamp of event occurrence or timestamp of event ingestion in Kafka, specified as a datetime scalar.

Data Types: `datetime`

Version History

Introduced in R2022b

See Also

`readtimetable` | `kafkaStream`

readtimetable

Package: `matlab.io.stream.event`

Read timetable from event stream

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
tt = readtimetable(stream)
```

Description

`tt = readtimetable(stream)` creates a timetable from an event stream.

`readtimetable` converts events from an event stream into rows of a timetable, where:

- The names in the event body become the timetable column names
- The value associated with each name becomes the column value in the event row.
- The event timestamp becomes the row timestamp.

Examples

Create Timetable from Event Stream

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Create a `KafkaStream` object for reading from and writing to the `CoolingFan` topic.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan");
```

Read events from the `CoolingFan` topic into a timetable.

```
tt = readtimetable(ks)
```

```
tt =

  50×11 timetable

         timestamp           vMotor    wMotor    Tmass 
    ____________________    ______    ______    ______

    31-Oct-2020 00:00:00    1.0909         0        25
    31-Oct-2020 00:00:00    1.1506     100.5     25.17
    31-Oct-2020 00:00:00    1.1739     190.9    25.223
    31-Oct-2020 00:00:00     1.162    267.96    25.192
    31-Oct-2020 00:00:00    1.1454    330.61     25.15
```

```
         :                :         :         :

31-Oct-2020 00:00:19    1.0269    239.35    25.785
31-Oct-2020 00:00:19    1.0332    240.45    25.803
31-Oct-2020 00:00:19    1.0267    263.98    25.784
31-Oct-2020 00:00:19    1.0262    243.69    25.783
31-Oct-2020 00:00:19    1.0262    257.21    25.783

Display all 50 rows.
```

Input Arguments

`stream` — Object connected to event stream
`KafkaStream` object | `InMemoryStream` object | `TestStream` object

Object connected to an event stream, specified as a `KafkaStream`, `InMemoryStream`, or `TestStream` object.

Version History

Introduced in R2022b

See Also

`writetimetable` | `kafkaStream` | `inMemoryStream` | `testStream` | `preview` | `seek`

Topics
"Process Kafka Events Using MATLAB" on page 11-5

seek

Package: `matlab.io.stream.event`

Set position in event stream to begin processing events

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
seek(esp,position)
seek(esp,position,Name=Value)
```

Description

`seek(esp,position)` sets the stream position at which to begin processing the event stream with processor `esp`.

`seek(esp,position,Name=Value)` additionally specifies options to manage data caching.

Examples

Process Events from Beginning of Stream

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `RecamanSequence`.

Create a `KafkaStream` object connected to the `RecamanSequence` topic.

```
ks = kafkaStream("kafka.host.com",9092,"RecamanSequence");
```

Assume that you have a stateful streaming analytic function `recamanSum` and a function to initialize the per-iteration state data called `initRecamanSum`. Create an `EventStreamProcessor` object that runs the `recamanSum` function and initializes the state data for the first iteration with the `initRecamanSum` function.

```
esp = eventStreamProcessor(ks,@recamanSum,@initRecamanSum);
```

```
esp = 

  EventStreamProcessor with properties:

      StreamFunction: @recamanSum
         InputStream: [1×1 matlab.io.stream.event.KafkaStream]
        OutputStream: [1×1 matlab.io.stream.event.InMemoryStream]
        InitialState: @initRecamanSum
       GroupVariable: [0×0 string]
        ReadPosition: Beginning
         ArchiveName: "recamanSum"
    ResetStateOnSeek: 1
```

Iterate the streaming analytic function over ten event windows.

```
execute(esp,10);
```

Check the result of the `recamanSum` function.

```
result = readtimetable(esp.OutputStream)
```

Input Arguments

esp — Object to process event streams
`EventStreamProcessor` object

Object to process event streams, specified as an `EventStreamProcessor` object.

`position` — Position in event stream
`"Beginning"` | `"End"` | `"Current"`

Position in an event stream, specified as one of the following values.

- `"Beginning"` — First event available in the event stream
- `"End"` — End of the event stream, which is one event past the latest event in the stream
- `"Current"` — Just past the current event in the stream

Example: `seek(esp,"Beginning")` moves the event stream position to the first event in the event stream.

Data Types: `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `seek(esp,"Beginning",ClearState=true)` moves the read position to the first event in the stream and clears the persistent data state.

`ClearState` — Flag to clear persistent state of data after calling seek
true | false

Flag to clear persistent state of data after calling the `seek` function, specified as a logical scalar. The default value is the value of the `EventStreamProcessor` property `ResetStateOnSeek`.

Data Types: `logical`

`PersistState` — Per-iteration state value of stream processing function
any valid MATLAB data type

Per-iteration state value of the stream processing function after calling the `seek` function, specified as any valid MATLAB data type.

If the streaming analytic function is stateful, then `PersistState` must be a MATLAB value of the same type as the value returned by the state initialization function set in `esp`. If the streaming analytic function is stateless, you cannot specify `PersistState`.

Version History
Introduced in R2022b

See Also
`eventStreamProcessor` | `seek` | `execute`

setProviderProperties

Package: `matlab.io.stream.event`

Set properties specific to Kafka configuration

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
setProviderProperties(ks,propname1,propval1,...,propnameN,propvalN)
```

Description

`setProviderProperties(ks,propname1,propval1,...,propnameN,propvalN)` sets Kafka stream provider properties on page 10-83.

Note It is recommended that you set Kafka provider properties when creating a Kafka stream object using `kafkaStream`. If you need to set properties after object creation using `setProviderProperties`, set them before the object interacts with the stream, such as when reading or writing data to a Kafka topic. After the stream object interacts with an event stream, setting Kafka properties might have no effect.

Examples

Set Kafka Security Protocol

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Create an event stream object connected to a Kafka topic.

```
ks = kafkaStream("kafka.host.com",9092,"coolingFan");
```

Set security properties that are specific to Kafka.

```
setProviderProperties(ks, ...
    "security.protocol","SASL_SSL", ...
    "sasl.mechanism","SCRAM-SHA-512");
```

Alternatively, you can set these properties when you create the object.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan", ...
     "security.protocol","SASL_SSL", ...
     "sasl.mechanism","SCRAM-SHA-512");
```

Input Arguments

`ks` — Object connected to Kafka stream topic

`KafkaStream` object

Object connected to a Kafka stream topic, specified as a `KafkaStream` object.

`propname` — Name of Kafka provider property

string scalar | character vector

Name of a Kafka provider property, specified as a string scalar or character vector.

Example: `"security.protocol"`

`propval` — Value of Kafka provider property

MATLAB expression

Value of a Kafka provider property, specified as a MATLAB expression. The expression must be a string or convertible to a string.

Example: `"SASL_SSL"`

More About

Stream Provider Properties

Stream providers such as Kafka have vendor-specific configuration properties that influence how the stream provider operates on the stream. Some properties apply to a single type of operation, such as topic creation, while other properties apply to multiple operations. Properties are grouped into categories so that properties are sent only to the provider with operations that support them. Because the same property can belong to multiple categories, each property is uniquely identified by the combination of its name and category.

Version History

Introduced in R2022b

See Also

`kafkaStream` | `getProviderProperties` | `isProperty`

Topics

"Connect to Secure Kafka Cluster" on page 11-9

seek

Package: `matlab.io.stream.event`

Set read position in event stream

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
offset = seek(stream,position)
offset = seek(stream,position,origin)
```

Description

`offset = seek(stream,position)` sets the read position of event stream `stream` to `position`. You can specify a numeric absolute position or a relative position, such as `"Beginning"` or `"End"` for the start or end of the stream, respectively. `seek` returns the position from which the next read operation occurs.

`offset = seek(stream,position,origin)` moves the read position of an event stream `position` number of events relative to the specified `origin`.

Examples

Seek to Beginning of Event Stream

Create a `TestStream` object to read from and write events to an event stream hosted by MATLAB.

```
ts = testStream;
```

Write sample timetable data to the event stream.

```
load indoors
writetimetable(ts,indoors)
```

Read data from the stream. The stream contains 60 events, but the stream by default has a window size of 100. Therefore, `readtimetable` reads the entire stream in a single read. The read position is one position past the end of the stream.

```
tt = readtimetable(ts);
```

Move the read position back to the first event.

```
seek(ts,"Beginning");
```

Seek to Position Relative to End of Event Stream

Create an `InMemoryStream` object to read from and write events to an event stream hosted by MATLAB. Configure the object to read 10 events at a time.

```
numRows = 10;
is = inMemoryStream(Rows=numRows);
```

Write timetable data to the event stream.

```
load indoors
writetimetable(is,indoors)
```

Move the read position 10 rows back from the end position.

```
seek(is,-numRows + 1,"End");
```

Read the last 10 events from the stream.

```
tt = readtimetable(is)
```

```
tt =

  10×2 timetable

            Time            Humidity    AirQuality
    ___________________    ________    __________

    2015-11-17 13:00:19       37           79
    2015-11-17 14:13:31       37           80
    2015-11-17 15:26:43       37           79
    2015-11-17 16:39:55       37           77
    2015-11-17 17:53:07       37           79
    2015-11-17 19:06:19       37           79
    2015-11-17 20:19:31       37           80
    2015-11-17 21:32:43       37           81
    2015-11-17 22:45:55       37           79
    2015-11-17 23:59:07       35           79
```

Input Arguments

`stream` — Object connected to event stream
`KafkaStream` object | `InMemoryStream` object | `TestStream` object

Object connected to an event stream, specified as a `KafkaStream`, `InMemoryStream`, or `TestStream` object.

`position` — Position in event stream
integer | datetime scalar | `"Beginning"` | `"End"` | `"Current"`

Position in an event stream at which the next read starts, specified as an absolute or relative position.

Absolute Position

Specify one of these values:

- A positive integer indicating the number of events from the start of the event stream that the read position moves to. You cannot specify an integer greater than the length of the stream. For example, `seek(stream,5)` moves the read position to the fifth event in the stream.
- A datetime scalar indicating the event timestamp that the read position moves to. If the stream contains no event corresponding to the specified date or time, the stream position moves to the first time after the specified time. For example, `seek(stream,datetime(2022,5,13,16,34,26))` moves the read position to the event that has a timestamp of `13-May-2022 16:34:26`.

Relative Position

Specify one of these values:

- `"Beginning"` — First event available in stream. For example: `seek(stream,"Beginning")`.
- `"End"` — Just past the last event in the stream. For example: `seek(stream,"End")`.
- `"Current"` — Just past the current event in the stream. For example: `seek(stream,"End")`.
- If you specify `origin`, a positive or negative integer indicating the number of events relative to that origin. For example, `seek(stream,-10,"End")` moves the read position 10 positions from the last event in the stream.

  `position` must not exceed either end of the stream relative to `origin`. If `origin` is `"End"`, then `position` must be negative. If `origin` is `"Beginning"`, then `position` must be positive.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `string` | `datetime`

`origin` — Origin that read position is relative to
`"Beginning"` | `"End"` | `"Current"`

Origin that the read position specified by `position` is relative to, specified as one of these values:

- `"Beginning"` — Read position is relative to the first event available in stream.
- `"End"` — Read position is relative to the last event in the stream.
- `"Current"` — Read position is relative to the current event in the stream.

Data Types: `string` | `char`

Version History
Introduced in R2022b

See Also
`kafkaStream` | `inMemoryStream` | `testStream` | `readtimetable` | `writetimetable` | `preview` | `seek`

setvartype

Package: `matlab.io.stream.event`

Set data types used to import and export variables to stream

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
opts = setvartype(opts,selection,type)
opts = setvartype(opts,selection,type)
```

Description

`opts = setvartype(opts,selection,type)` sets the data types of all variables in the object `opts` to the data type specified by `type`.

- If `opts` is an `ImportOptions` object, then `setvartype` returns an `ImportOptions` object.
- If `opts` is an `ExportOptions` object, then `setvartype` returns an `ExportOptions` object.

`opts = setvartype(opts,selection,type)` sets the data types of the variables specified by `selection` to the data types specified by `type` in the object `opts`.

Examples

Set Data Type of Imported Variables in Event Stream

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `CoolingFan`.

Create an object to connect to Kafka streaming data.

```
ks = kafkaStream("kafka.host.com",9092,"CoolingFan")
```

Create an import options object from the `KafkaStream` object.

```
io = detectImportOptions(ks)
```

```
io = 

  ImportOptions with properties:

            VariableNames: ["vMotor"    "wMotor"    "Tmass"    …    ]
            VariableTypes: ["double"    "double"    "double"    …    ]
              KeyVariable: "key"
    SelectedVariableNames: ["vMotor"    "wMotor"    "Tmass"    …    ]
```

Examine data types of the variables.

```
disp([io.VariableNames' io.VariableTypes'])
```

```
    "vMotor"                  "double"
    "wMotor"                  "double"
    "Tmass"                   "double"
    "ExternalTempAnomaly"     "double"
    "FanDragAnomaly"          "double"
    "VoltageSourceAnomaly"    "double"
    "FanRow"                  "double"
    "FanColumn"               "double"
    "FanID"                   "double"
    "GroupID"                 "double"
    "key"                     "string"
```

Change the data types of `vMotor` and `wMotor` variables to `int32`.

```
io = setvartype(io,{"vMotor","wMotor"},"int32")
```

```
io = 

  ImportOptions with properties:

            VariableNames: ["vMotor"    "wMotor"    "Tmass"    …    ]
            VariableTypes: ["int32"    "int32"    "double"    …    ]
              KeyVariable: "key"
    SelectedVariableNames: ["vMotor"    "wMotor"    "Tmass"    …    ]
```

Import the variables with their updated types using `readtimetable`.

```
tt = readtimetable(ks,io);
```

Alternatively, you can set the `ImportOptions` property of the stream object and the use `readtimetable`.

```
ks.ImportOptions = io;
tt = readtimetable(ks);
```

Set Data Type of Exported Variables in Event Stream

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has the topics `Triangles` and `numericTriangles`.

Create a `KafkaStream` object connected to the `Triangles` topic.

```
inKS = kafkaStream("kafka.host.com",9092,"Triangles");
```

Read events from the `Triangles` topic into a timetable. Preview the data by viewing the first row. The `a`, `b`, and `c` triangle side lengths are stored as strings.

```
tt = readtimetable(inKS);
row = tt(1,:)
```

```
row = 

  1×3 timetable

     timestamp       a       b       c  
    ___________    _____   _____   _____
    03-Sep-2022    "15"    "31"    "36"
```

Use `detectExportOptions` to generate an `ExportOptions` object from the Kafka stream object. The function obtains the types used to export the variables from the first row of the timetable.

```
opts = detectExportOptions(inKS,row);
```

Use `getvartype` to confirm that the side length variables are currently exported to the stream as strings.

```
type = getvartype(opts,["a" "b" "c"]);
```

```
type = 

  1×3 string array

    "string"    "string"    "string"
```

Update the export options so that the side lengths are exported as `double` values. Confirm the updated options by using `getvartype`.

```
opts = setvartype(opts,["a","b","c"],"double");

[name,type] = getvartype(opts);
fprintf("%s: %s\n", [name; type])
```

```
a: double
b: double
c: double
```

Connect to the stream to export data to `numericTriangles`.

```
outKS = kafkaStream("kafka.host.com",9092,"numericTriangles", ...
    ExportOptions=opts)
```

```
outKS = 

  KafkaStream with properties:

                  Topic: "numericTriangles"
                  Group: "85c42e39-695d-467a-86f0-f0095792e7de"
                  Order: EventTime
                   Host: "kafka.host.com"
                   Port: 9092
      ConnectionTimeout: 30
         RequestTimeout: 61
          ImportOptions: "None"
          ExportOptions: "Source: string"
          PublishSchema: "true"
             WindowSize: 50
            KeyVariable: "key"
            KeyEncoding: "utf16"
                KeyType: "text"
           KeyByteOrder: "BigEndian"
           BodyEncoding: "utf8"
             BodyFormat: "JSON"
              ReadLimit: "Size"
    TimestampResolution: "Milliseconds"
```

Export the timetable to the new stream. The triangle side lengths in this stream are of type `double`.

```
writetimetable(outKS,tt);
```

Input Arguments

`opts` — Event stream options
`ImportOptions` object | `ExportOptions` object

Event stream options, specified as an `ImportOptions` or `ExportOptions` object. The `opts` object contains properties that control the data import/export process, such as variable names and types.

`selection` — Selected variables
character vector | string scalar | cell array of character vectors | string array

Selected variables, specified as a character vector, string scalar, cell array of character vectors, or string array.

Variable names must be a subset of the names recognized by the `opts` object.

Example: `'FanID'`

Example: `"FanID"`

Example: `{'FanID','vMotor'}`

Example: `["FanID" "vMotor"]`

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

`type` — New data type of variable
string scalar

New data type of variable, specified as a string scalar containing a valid MATLAB data type name. The variable `type` designates the data type to use when importing or exporting the variable. Use one of the data types listed in this table.

Data	MATLAB Data Type
Text	`"char"` or `"string"`

Data	MATLAB Data Type
Numeric	`"single"`, `"double"`, `"int8"`, `"int16"`, `"int32"`, `"int64"`, `"uint8"`, `"uint16"`, `"uint32"`, or `"uint64"` Undefined floating-point numbers `NaN`, `-Inf`, `+Inf` are only valid for `single` and `double` data types. Therefore, when you change the type of floating-point data to an integer, the importing/exporting function converts the undefined floating-point numbers to valid integers. For example, when converting to the `"uint8"` data type: • `NaN` is converted to `0`. • `-Inf` is converted to `intmin("int8")`. • `+Inf` is converted to `intmax("int8")`. The same conversion process applies to all the integer data types: `int8`, `int16`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, or `uint64`.
Logical	`"logical"`

Example: `io = setvartype(io,"vMotor","int32")` changes the data type of the event stream variable `vMotor` to `int32`.

Data Types: `string`

Version History

Introduced in R2022b

See Also

`getvartype` | `detectImportOptions` | `detectExportOptions`

start

Package: `matlab.io.stream.event`

Start processing event streams using local test server

Note This function requires Streaming Data Framework for MATLAB® Production Server™ and MATLAB Compiler SDK™.

Syntax

```
start(esp)
start(esp,port,host)
```

Description

`start(esp)` starts processing event streams using a local test server (development version of MATLAB Production Server) running at the default hostname `localhost` and port number 9910. Asynchronous event processing started with the `start` function continues until either the processor reaches the end of the stream or you explicitly call `stop`.

Note Before starting event processing with `start`, you must start the local test server with `startServer`.

`start(esp,port,host)` specifies the port number and the hostname of the machine on which the local test server is running.

Examples

Start Processing Event Streams Using Local Test Server

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `RecamanSequence`.

Also assume that you have a stateful streaming analytic function `recamanSum` and initialization function `initRecamanSum`.

Create a `KafkaStream` object connected to the `RecamanSequence` topic.

```
ks = kafkaStream("kafka.host.com",9092,"RecamanSequence");
```

Create an `EventStreamProcessor` object that runs the `recamanSum` function and is initialized by the `initRecamanSum` function.

```
esp = eventStreamProcessor(ks,@recamanSum,@initRecamanSum);
```

Start the local test server, which also opens the **Production Server Compiler** app.

```
startServer(esp);
```

Start the test server from the app by clicking **Test Client** and then **Start**. For an example on how to use the app, see “Test Client Data Integration Against MATLAB” (MATLAB Compiler SDK).

Navigate back to the MATLAB command prompt to start processing events.

```
start(esp);
```

In the **Production Server Compiler** app, the test server receives data.

After you finish testing the processing of events, use the `stop` function to stop event processing and the `stopServer` function to shut down the server.

Input Arguments

`esp` — Object to process event streams
`EventStreamProcessor` object

Object to process event streams, specified as an `EventStreamProcessor` object.

`port` — Port number on which test server is running
integer in range [0, 65,535]

Port number on which the test server is running, specified as an integer in the range [0, 65,535].

Example: `9920`

`host` — Hostname of machine on which local test server is running
string | character vector

Hostname of machine on which the local test server is running, specified as a string or character vector.

Example: `'144.213.5.7'` or `'localhost'`

Data Types: `string` | `char`

Version History

Introduced in R2022b

See Also

`stop` | `startServer` | `stopServer` | `eventStreamProcessor`

Topics
“Test Streaming Analytic Function Using Local Test Server” on page 11-12

startServer

Package: `matlab.io.stream.event`

Start local test server

Note This function requires Streaming Data Framework for MATLAB® Production Server™ and MATLAB Compiler SDK™.

Syntax

```
startServer(esp)
startServer(esp,ExtraFiles=files)
```

Description

`startServer(esp)` launches a local test server (development version of MATLAB Production Server) that simulates the production environment so that you can test event processing.

`startServer` generates a MATLAB project file for the **Production Server Compiler** app. In addition to simulating production with this file on a local test server, you can use the generated project file to create a CTF archive.

`startServer(esp,ExtraFiles=files)` adds additional files to the CTF archive when starting the local test server.

Examples

Process Event Streams Using Local Test Server

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `RecamanSequence`.

Also assume that you have a streaming analytic function `recamanSum` and a function `initRecamanSum` to initialize persistent state.

Create a `KafkaStream` object connected to the `RecamanSequence` topic.

```
ks = kafkaStream("kafka.host.com",9092,"RecamanSequence");
```

Create an `EventStreamProcessor` object that runs the `recamanSum` function, which is initialized by the `initRecamanSum` function.

```
esp = eventStreamProcessor(ks,@recamanSum,@initRecamanSum);
```

Start the local test server, which also opens the **Production Server Compiler** app.

Note To use the test server, you require MATLAB Compiler SDK.

```
startServer(esp);
```

Once the app opens, you must start the test server manually.

To start the test server from the app, click **Test Client** and then **Start**. For an example on how to use the app, see “Test Client Data Integration Against MATLAB” (MATLAB Compiler SDK).

Navigate back to the MATLAB command prompt to start processing events.

```
start(esp);
```

Using the MATLAB editor, you can set breakpoints in the `recamanSum` function to examine the incoming streaming data when you start the server.

Input Arguments

esp — Object to process event streams
`EventStreamProcessor` object

Object to process event streams, specified as an `EventStreamProcessor` object.

`files` — Additional files to include in archive
character vector | string scalar | string array

Additional files to include in the generated archive, specified as a character vector or string scalar for a single file, or as a string array for multiple files.

Extra files are necessary only if you plan to use the generated project file to deploy a CTF archive to MATLAB Production Server. For other ways to create deployable archives, see the `package` and `streamingDataCompiler` functions.

Example: `archive = startServer(esp,"ExtraFiles"=["data.mat", "/schema/registry/schema.json"])` includes the files `data.mat` and `schema.json` in the generated deployable archive.

Data Types: `char` | `string`

Version History

Introduced in R2022b

See Also

`eventStreamProcessor` | `stopServer` | `start` | `stop`

Topics

“Test Streaming Analytic Function Using Local Test Server” on page 11-12

streamingDataCompiler

Package stream processing function into deployable archive

Note This function requires Streaming Data Framework for MATLAB® Production Server™ and MATLAB Compiler SDK™.

Syntax

```
filePath = streamingDataCompiler(streamFcn,inStream,outStream)
filePath = streamingDataCompiler(streamFcn,inStream,outStream,Name=Value)
```

Description

`filePath = streamingDataCompiler(streamFcn,inStream,outStream)` creates a MATLAB Compiler SDK project file and opens the **Production Server Compiler** app. Use this app to package the streaming function into a deployable archive for MATLAB Production Server.

`streamingDataCompiler` returns the full path to the project file.

`filePath = streamingDataCompiler(streamFcn,inStream,outStream,Name=Value)` sets additional options for packaging the function.

For example, if you specify `OutputType="Archive"`, the `streamingDataCompiler` function returns a deployable archive (CTF file) instead of a project file.

Examples

Package Streaming Analytic Function for Deployment to MATLAB Production Server

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `RecamanSequence`.

Also assume that you have a streaming analytic function `recamanSum` and a function `initRecamanSum` to initialize persistent state.

Create a `KafkaStream` object connected to the `RecamanSequence` topic.

```
ks = kafkaStream("kafka.host.com",9092,"RecamanSequence");
```

Create another `KafkaStream` object to write the results of the streaming analytic function to a different topic called `RecamanSequenceResults`.

```
outks = kafkaStream("kafka.host.com",9092,"RecamanSequenceResults");
```

Package the streaming analytic function `recamanSum` into a deployable archive. Since the analytic function uses the state initialization function `initRecamanSum`, also specify the `StateStore` property as an input argument.

```
file = streamingDataCompiler(@recamanSum,ks,outKS, ...
    StateStore="KVStore",InitialState=@initRecamanSum)
```

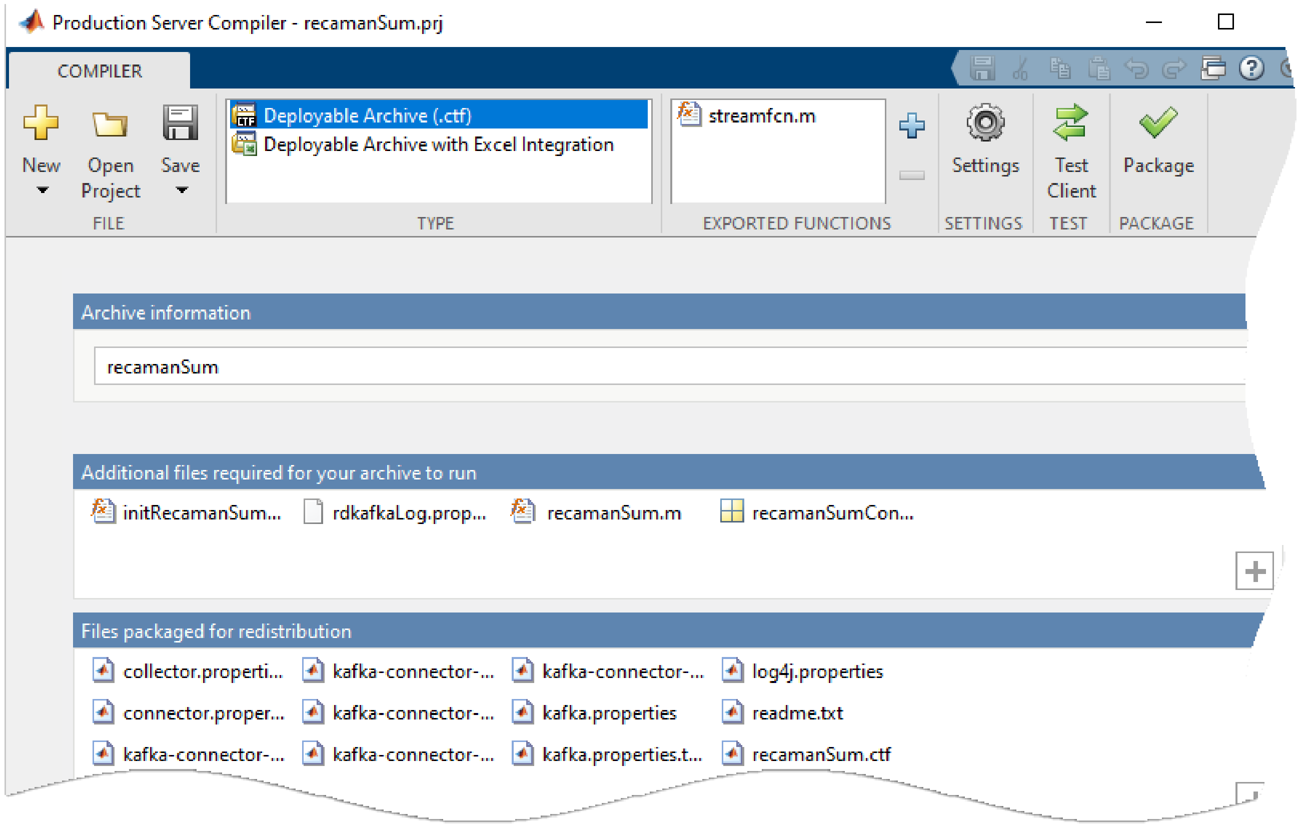
```
file = 

    "J:\recamanSum.prj"
```

The `package` function generates a MATLAB project file based on the `eventStreamProcessor` object, returns the path to this file, and opens this project file with the **Production Server Compiler** app. The project file contains values for:

- The streaming analytic function, `recamanSum.m`
- The entry point function, `streamfcn.m`
- The deployable archive, `RecamanSum.ctf`

To modify the list of deployed functions or the name of the generated archive, see “Customize Application and Its Appearance” (MATLAB Compiler SDK).



In the **Production Server Compiler**, click **Package** to generate the deployable archive. You can deploy the generated archive to MATLAB Production Server. For more information on deploying to MATLAB Production Server, see “Deploy Archive to MATLAB Production Server”.

Input Arguments

`streamFcn` — Streaming analytic function

function handle | string | character vector

Streaming analytic function, specified as a function handle, string, or character vector.

Data Types: `function_handle` | `string` | `char`

`inStream` — Event stream from which streaming analytic function reads events
stream connector object

Event stream from which the streaming analytic function reads events, specified as a stream connector object, such as `KafkaStream` or `TestStream`.

`outStream` — Event stream to which streaming analytic function writes events
stream connector object

Event stream from which the streaming analytic function reads events, specified as a stream connector object, such as `KafkaStream` or `TestStream`. `InMemoryStream` objects are not supported.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `streamingDataCompiler(streamFcn,inStream,outStream,OutputFolder='J:\')`

Compile Configuration Options

`ArchiveName` — Name of generated deployable archive
`streamFcn` (default) | string | character vector

Name of the generated deployable archive, specified as a string or character vector.

Data Types: `char` | `string`

`OutputType` — File type
`Project` (default) | `Archive`

Type of file created by the streamingDataCompiler function, specified as one of these values.

- `Project` — Creates a `productionServerCompiler` project file and launches the **Production Server Compiler** app. You can create a CTF file using the **Production Server Compiler** app.
- `Archive` — Creates a CTF file.

Data Types: `char` | `string`

`ExtraFiles` — Additional files to include in archive
character vector | string scalar | string array

Additional files to include in the generated archive, specified as a character vector or string scalar for a single file or as a string array for multiple files.

Example: `ExtraFiles=["data.mat","/schema/registry/schema.json"]` includes the files `data.mat` and `schema.json` in the generated deployable archive.

Data Types: `char` | `string`

`OutputFolder` — Location of generated file
current folder (default) | string | character vector

Location of the generated file, specified as a string or character vector.

Example: `OutputFolder='J:\'` saves the generated file in `J:\`.

Data Types: `string` | `char`

`OpenProject` — Flag to automatically open project in MATLAB
`true` (default) | `false`

Flag to automatically open the project in MATLAB, specified as logical `true` or `false`. This property is incompatible with `OutputType="Archive"`.

Data Types: `logical`

Execution Configuration Options

`InitialState` — Function that creates initial state for streaming analytic function
function handle | string | character vector

Function that creates the initial state for the streaming analytic function, specified as either a function handle, string, or character vector.

Data Types: `function_handle` | `char` | `string`

`StateStore` — Persistent storage connection name
string | character vector

Persistent storage connection name, specified as a string or character vector. You must specify a `StateStore` when using `InitialState` or when using a stateful stream function. The connection name must be known to the MATLAB Production Server instance to which the archive will be deployed. For more information on using a data cache for persistent storage, see “Data Caching Basics”.

Data Types: `char` | `string`

`GroupByKey` — Flag to call streaming function with events that have same key
`false` (default) | `true`

Flag to call the streaming function with events that have same key, specified as a logical `true` or `false`. Setting this flag to `true` splits an event window into subwindows with homogeneous keys and calls the streaming function once per subwindows.

Data Types: `logical`

Version History

Introduced in R2022b

See Also

`eventStreamProcessor` | `package`

Topics

“Deploy Streaming Analytic Function to MATLAB Production Server” on page 11-17

stop

Package: `matlab.io.stream.event`

Stop processing event streams using local test server

Note This function requires Streaming Data Framework for MATLAB® Production Server™ and MATLAB Compiler SDK™.

Syntax

`stop(esp)`

Description

`stop(esp)` stops processing event streams using a local test server (development version of MATLAB Production Server). Asynchronous event processing started with the `start` function continues until either the processor reaches the end of the stream or there is an explicit call to the `stop` function.

Examples

Stop Processing Event Streams

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `RecamanSequence`.

Also assume that you have a stateful streaming analytic function `recamanSum` and initialization function `initRecamanSum`.

Create a `KafkaStream` object connected to the `RecamanSequence` topic.

```
ks = kafkaStream("kafka.host.com",9092,"RecamanSequence");
```

Create an `EventStreamProcessor` object that runs the `recamanSum` function.

```
esp = eventStreamProcessor(ks,@recamanSum,@initRecamanSum);
```

Start the test server, which also opens the **Production Server Compiler** app.

```
startServer(esp);
```

To start the test server from the app, click **Test Client** and then **Start**. For an example on how to use the app, see “Test Client Data Integration Against MATLAB” (MATLAB Compiler SDK).

Navigate back to the MATLAB command prompt to start processing events.

```
start(esp);
```

In the **Production Server Compiler** app, the test server receives data.

From the MATLAB command prompt, stop the event processing.

```
stop(esp);
```

Then, you can shut down the server using the `stopServer` function or by clicking **Stop** in the app UI.

Input Arguments

esp — Object to process event streams
`EventStreamProcessor` object

Object to process event streams, specified as an `EventStreamProcessor` object.

Version History

Introduced in R2022b

See Also

`start` | `stopServer` | `eventStreamProcessor` | `startServer`

Topics
"Test Streaming Analytic Function Using Local Test Server" on page 11-12

stop

Package: `matlab.io.stream.event`

Stop processing event streams from Kafka topic

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
stop(ks)
```

Description

`stop(ks)` stops processing event stream data from a Kafka topic and shuts down any external processes interacting with the stream.

Examples

Stop Streaming Data from Kafka Topic

Assume that you have a Kafka server running at the network address `kafka.host.com:9092`. This server has a topic `RecamanSequence` that contains numbers in Recamán's sequence.

Create a `KafkaStream` object for reading event stream data from the `RecamanSequence` topic.

```
ks = kafkaStream("kafka.host.com",9092,"RecamanSequence");
```

Read the first 50 rows from the stream.

```
tt = readtimetable(ks)
```

```
tt =

  50×2 timetable

         timestamp          R     key 
    ____________________    __    _____

    27-Jun-2022 18:37:52     0    "0"
    27-Jun-2022 18:37:53     1    "1"
    27-Jun-2022 18:37:54     3    "2"
    27-Jun-2022 18:37:55     6    "3"
    27-Jun-2022 18:37:56     2    "4"
    27-Jun-2022 18:37:57     7    "5"
    27-Jun-2022 18:37:58    13    "6"
    27-Jun-2022 18:37:59    20    "7"

             :              :      :

    27-Jun-2022 18:38:34    37    "42"
    27-Jun-2022 18:38:35    80    "43"
    27-Jun-2022 18:38:36    36    "44"
    27-Jun-2022 18:38:37    81    "45"
    27-Jun-2022 18:38:38    35    "46"
    27-Jun-2022 18:38:39    82    "47"
    27-Jun-2022 18:38:40    34    "48"
    27-Jun-2022 18:38:41    83    "49"
```

Preview the next 8 rows in the stream. The read position is set immediately after the last row read from the stream.

```
preview(ks)
```

```
ans =

  8×2 timetable

         timestamp          R     key 
    ____________________    __    ____

    27-Jun-2022 18:38:42    33    "50"
    27-Jun-2022 18:38:43    84    "51"
    27-Jun-2022 18:38:44    32    "52"
    27-Jun-2022 18:38:45    85    "53"
    27-Jun-2022 18:38:46    31    "54"
    27-Jun-2022 18:38:47    86    "55"
    27-Jun-2022 18:38:48    30    "56"
    27-Jun-2022 18:38:49    87    "57"
```

Stop processing the stream.

```
stop(ks)
```

Input Arguments

ks — Object connected to Kafka stream topic
`KafkaStream` object

Object connected to a Kafka stream topic, specified as a `KafkaStream` object.

Version History

Introduced in R2022b

See Also

`kafkaStream` | `readtimetable` | `writetimetable` | `preview` | `seek`

Topics
"Test Streaming Analytic Function Using Local Test Server" on page 11-12

stopServer

Package: `matlab.io.stream.event`

Shut down local test server

Note This function requires Streaming Data Framework for MATLAB® Production Server™ and MATLAB Compiler SDK™.

Syntax

```
stopServer(esp)
```

Description

`stopServer(esp)` shuts down the local test server (development version of MATLAB Production Server) used to simulate event processing in a production environment.

Examples

Stop Local Test Server Used to Process Events

Assume that you have a Kafka server running at the network address `kafka.host.com:9092` that has a topic `RecamanSequence`.

Also assume that you have a stateful streaming analytic function `recamanSum` and initialization function `initRecamanSum`.

Create a `KafkaStream` object connected to the `RecamanSequence` topic.

```
ks = kafkaStream("kafka.host.com",9092,"RecamanSequence");
```

Create an `EventStreamProcessor` object that runs the `recamanSum` function, which is initialized by the `initRecamanSum` function.

```
esp = eventStreamProcessor(ks,@recamanSum,@initRecamanSum);
```

Start the local test server, which also opens the development version of MATLAB Production Server in the **Production Server Compiler** app.

```
startServer(esp);
```

You can then use the `start` and `stop` functions to start and stop event processing, respectively.

`stop` causes the client of the `EventStreamProcessor` object to stop sending events from the stream to the local test server, but it does not shut down the server. After you finish testing the processing of events, stop the test server by calling `stopServer`.

```
stopServer(esp);
```

Input Arguments

esp — Object to process event streams
`EventStreamProcessor` object

Object to process event streams, specified as an `EventStreamProcessor` object.

Version History

Introduced in R2022b

See Also

`startServer` | `start` | `stop` | `eventStreamProcessor` | `execute`

Topics
“Test Streaming Analytic Function Using Local Test Server” on page 11-12

testStream

Create connection to event stream hosted by MATLAB with schema processing applied

Note This object requires Streaming Data Framework for MATLAB® Production Server™.

Description

The `testStream` function creates a `TestStream` object, which you can use to test reading from and writing to event streams hosted by MATLAB. `TestStream` objects store events in MATLAB variables, which are exported to event streams based on the data type values specified by the `ExportOptions` property. `TestStream` applies the standard configurable schema processing when reading and writing timetable data. Use this object to test your schema management before deployment, without requiring a streaming service host. The data in a `TestStream` object disappears when you exit MATLAB.

Use `TestStream` to develop your streaming analytic function without having to connect to or consume resources from a streaming service such as Kafka. `TestStream` stores data using MATLAB memory space, so use this object with finite-size data sets. Because the data is stored in MATLAB, `TestStream` objects typically stream data faster than streams that have data stored on a network or file system.

Creation

Syntax

```
ts = testStream
ts = testStream(Rows=numevents)

ts = testStream(Duration=timespan)

ts = testStream( ___ ,Name=Value)
```

Description

Row-Based Event Window

`ts = testStream` creates a default `TestStream` object connected to an event stream hosted by MATLAB. The object reads 100 stream event rows at a time.

`ts = testStream(Rows=numevents)` creates a `TestStream` object that reads `numevents` stream event rows at a time.

Duration-Based Event Window

`ts = testStream(Duration=timespan)` creates a `TestStream` object that reads stream events occurring during the specified timestamp span, `timespan`.

Additional Options

`ts = testStream( ___ ,Name=Value)` sets event stream properties on page 10-107 using one or more name-value arguments and any of the previous syntaxes.

Input Arguments

`numevents` — Number of events in event window
100 (default) | positive integer

Number of events in the event window, specified as a positive integer. `Rows=numevents` specifies the number of rows that a call to the `readtimetable` function returns. If there are less than the number of specified rows available for reading, then `readtimetable` times out and returns an empty timetable.

Example: `Rows=500` specifies that each call to `readtimetable` returns a timetable with 500 rows.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`timespan` — Timestamp span in event window
0 (default) | duration scalar

Timestamp span in the event window, specified as a duration scalar. `Duration=timespan` determines the events that the `readtimetable` function returns based on their timestamp. `timespan` specifies the difference between the last and first timestamps of events in the event window.

Example: `Duration=minutes(1)` specifies that each call to `readtimetable` returns a timetable that has one minute's worth of events, where the timestamp of the last event is no more than one minute later than the timestamp of the first event.

Data Types: `duration`

Properties

`Name` — Event stream name
string scalar | character vector

Name of the event stream, specified as a string scalar or character vector. This property is provided for compatibility with other stream objects. If you do not specify a name, the `testStream` function generates one when you create the `TestStream` object.

Example: `CoolingFan`

Data Types: `string` | `char`

`WindowSize` — Event window size
100 (default) | duration scalar | positive integer

This property is read-only.

Event window size, specified by a fixed amount of time (using the `Duration` argument) or a fixed number of messages (using the `Rows` argument).

Data Types: `duration` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

ReadLimit — Wait strategy
"Size" (default) | "Time"

Strategy to wait for a response from the stream, specified as one of these values:

- "Size" — Client prioritizes filling the event window. Using this strategy, the client might wait longer than the RequestTimeout time period as long as it is still receiving the expected number of messages. The default number of messages is 50. If the client receives no messages within the RequestTimeout time period, it no longer waits.
- "Time" — Client strictly adheres to the RequestTimeout limit, even if it has not received the expected number of messages. RequestTimeout specifies the amount of time the stream object waits between receiving events. If the stream is actively receiving data, it does not time out during that operation.

Note This object does not implement the ReadLimit property and does not have a RequestTimeout property. It is provided for compatibility with other stream connector objects. By setting the wait strategy in this object, you can more easily update your code to switch between objects that do implement this property, such as KafkaStream.

TimestampResolution — Unit of event timestamp
"Milliseconds" (default) | "Seconds" | "Minutes" | "Hours" | "Days"

Unit of event timestamp, specified as one of these values:

- "Milliseconds"
- "Seconds"
- "Minutes"
- "Hours"
- "Days"

Interpret the event timestamp as the number of corresponding units before or after the Unix epoch.

Data Types: string | char

Import and Export Options

ImportOptions — Rules for transforming stream events into MATLAB data
ImportOptions object

Rules for transforming stream events into MATLAB data, specified as an ImportOptions object. This object controls the import of stream events into MATLAB.

ExportOptions — Rules for transforming MATLAB data into stream events
ExportOptions object

Rules for transforming MATLAB data into stream events, specified as an ExportOptions object. This object controls the export of MATLAB data into streams.

PublishSchema — Flag to indicate whether export schema is written to output stream
true (default) | false

Flag to indicate whether the export schema is written to the output stream, specified as a logical scalar.

The schema is embedded in each event, which can significantly increase the size of the event. If downstream applications do not require the schema, set this flag to false to reduce the number of bytes in your stream.

Data Types: `logical`

Event Key and Body Encoding

`KeyVariable` — Name of key variable
`key` (default) | string scalar | character vector

Name of the key variable in the event stream, specified as a string scalar or character vector.

Data Types: `string` | `char`

`KeyEncoding` — Character encoding format for bits in event key
`uint8` (default) | `utf16` | `utf8` | `base64`

Character encoding format used to interpret the bits in an event key, specified as one of the following:

- `utf8` — UTF-8 encoding format
- `utf16` — UTF-16 encoding format
- `base64`— Base 64 encoding format
- `uint8` — Eight-bit unsigned binary bytes

If `KeyEncoding` is `utf8` or `utf16`, then the `KeyType` property must be `text`. If `KeyEncoding` is `base64` or `uint8`, then `KeyType` must be one of the numeric encoding formats.

KeyType — Character encoding scheme for bytes in event key
`text` (default) | `utf16` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `int64` | `uint64` | `single` | `double`

Character encoding scheme used to interpret the bytes in an event key, specified as one of these values:

- `uint8` — One-byte unsigned integer
- `int8` — One-byte signed integer
- `uint16` — Two-byte unsigned integer
- `int16` — Two-byte signed integer
- `uint32` — Four-byte unsigned integer
- `int32` — Four-byte signed integer
- `uint64` — Eight-byte unsigned integer
- `int64` — Eight-byte signed integer
- `single` — Single-precision IEEE 754 floating point number
- `double` — Double-precision IEEE 754 floating point number
- `text` — String

If `KeyType` is `text`, then the `KeyEncoding` property must be either `utf8` or `utf16`. If `KeyType` is any of the other numeric encoding formats, then `KeyEncoding` must be either `base64` or `uint8`.

`KeyByteOrder` — Order for storing bits in event key
`BigEndian` (default) | `LittleEndian` | `MatchHost` | `NotApplicable`

Order for storing bits in the event key, specified as one of the following.

- `LittleEndian` — Least significant bit is stored first
- `BigEndian` — Most significant bit is stored first
- `MatchHost`— Bits are stored in the same order as is used by the host computer on which the streaming data framework is running
- `NotApplicable` — Not an integer key

This property is applicable only for integer keys and not applicable to floating point or text keys.

`BodyEncoding` — Character encoding format for bits in event body
`utf8` (default) | `uint8` | `utf16` | `base64`

Character encoding format used to interpret the bits in the event body, specified as one of the following:

- `utf8` — UTF-8 encoding format
- `utf16` — UTF-16 encoding format
- `base64`— Base 64 encoding format
- `uint8` — Eight-bit unsigned binary bytes

This property determines the size and encoding of the bytes used in the event body, which are in the format specified by `BodyFormat`.

`BodyFormat` — Format of bytes in event body
`JSON` (default) | `Array` | `Text` | `Binary`

Format of bytes in event body, specified as one of the following:

- `JSON` — JSON string
- `Array` — MATLAB array
- `Text` — String data
- `Binary` — Binary data

Depending on the encoding specified by `BodyEncoding`, bytes can be larger than eight bits.

Object Functions

readtimetable	Read timetable from event stream
writetimetable	Write timetable to event stream
seek	Set read position in event stream
preview	Preview subset of events from event stream
identifyingName	Event stream name
detectImportOptions	Create import options based on event stream content
detectExportOptions	Create export options based on event stream content

Examples

Write and Preview Events for Test Stream

Create a `TestStream` object to read from and write events to an event stream hosted by MATLAB.

```
ts = testStream
```

```
t =

  TestStream with properties:

                 Name: "8ee84682-dcb6-4dc7-a10a-873f80dc5f98"
        ImportOptions: "None"
        ExportOptions: "Source: function eventSchema"
        PublishSchema: "true"
           WindowSize: 100
          KeyVariable: "key"
          KeyEncoding: "uint8"
              KeyType: "text"
         KeyByteOrder: "BigEndian"
         BodyEncoding: "utf8"
           BodyFormat: "JSON"
            ReadLimit: "Size"
  TimestampResolution: "Milliseconds"
```

Write sample timetable data to the event stream.

```
load indoors
writetimetable(ts,indoors)
```

Preview data from the timetable. The timetable data has no key column, but `writetimetable` generates an empty key column in the stream by default. This makes it easier to transition your code from using a `TestStream` object to a `KafkaStream` object, which includes this key column to identify the event source.

```
preview(ts)
```

```
ans =

  8×3 timetable

         timestamp          Humidity    AirQuality    key
    ____________________    ________    __________    ___

    15-Nov-2015 00:00:24       36           80        ""
    15-Nov-2015 01:13:35       36           80        ""
    15-Nov-2015 02:26:47       37           79        ""
    15-Nov-2015 03:39:59       37           82        ""
    15-Nov-2015 04:53:11       36           80        ""
    15-Nov-2015 06:06:23       36           80        ""
    15-Nov-2015 07:19:35       36           80        ""
    15-Nov-2015 08:32:47       37           80        ""
```

Version History

Introduced in R2022b

See Also

`kafkaStream` | `inMemoryStream`

Topics

"Streaming Data Framework for MATLAB Production Server Basics" on page 11-2

writetimetable

Package: `matlab.io.stream.event`

Write timetable to event stream

Note This function requires Streaming Data Framework for MATLAB® Production Server™.

Syntax

```
writetimetable(stream,tt)
writetimetable(ks,tt,MissingTopic=action)
```

Description

`writetimetable(stream,tt)` writes the timetable `tt` to the end of the event stream `stream`.

`writetimetable` converts rows of a timetable into events in an event stream, where:

- The column names in the timetable become variable names in the event body.
- The values in each event row become the values of those variables.
- The row timestamp becomes the event timestamp.

You can append data to a stream but cannot modify data that is already written to a stream.

`writetimetable(ks,tt,MissingTopic=action)` specifies whether to create a topic when writing a timetable to an event stream hosted by Kafka or fail the write operation when the topic is missing. If the Kafka cluster that you are writing to is configured to auto-create topics, specifying `action` has no effect.

Examples

Write Timetable to Event Stream

Load air quality data and weather measurements into a timetable.

```
load indoors
```

Create an `InMemoryStream` object to connect to an event stream hosted by MATLAB.

```
i = inMemoryStream;
```

Write the timetable to the event stream.

```
writetimetable(i,indoors)
```

Preview the data in the event stream.

```
preview(i)
```

```
ans =

  8×2 timetable

           Time            Humidity    AirQuality
    ___________________    ________    __________

    2015-11-15 00:00:24       36           80
    2015-11-15 01:13:35       36           80
    2015-11-15 02:26:47       37           79
    2015-11-15 03:39:59       37           82
    2015-11-15 04:53:11       36           80
    2015-11-15 06:06:23       36           80
    2015-11-15 07:19:35       36           80
    2015-11-15 08:32:47       37           80
```

Write Timetable to Stream Hosted by Kafka

Load air quality data and weather measurements into a timetable.

```
load indoors
```

Assume that you have a Kafka host running at network address `kafka.host.com:9092`. Create a `KafkaStream` object that processes 10 stream events at a time.

```
ks = kafkaStream("kafka.host.com",9092,"IndoorTemp",Rows=10);
```

Create the `IndoorTemp` topic and write the timetable to it.

```
writetimetable(ks,indoors)
```

Read the first 10 events from the stream.

```
tt1 = readtimetable(ks)
```

```
tt1 =

  10×3 timetable

         timestamp          Humidity    AirQuality    key
    ____________________    ________    __________    ___

    15-Nov-2015 00:00:24       36           80        ""
    15-Nov-2015 01:13:35       36           80        ""
    15-Nov-2015 02:26:47       37           79        ""
    15-Nov-2015 03:39:59       37           82        ""
    15-Nov-2015 04:53:11       36           80        ""
    15-Nov-2015 06:06:23       36           80        ""
    15-Nov-2015 07:19:35       36           80        ""
    15-Nov-2015 08:32:47       37           80        ""
    15-Nov-2015 09:45:59       37           79        ""
    15-Nov-2015 10:59:11       36           80        ""
```

Read the next 10 events from the stream.

```
tt2 = readtimetable(ks)
```

```
tt2 =

  10×3 timetable

         timestamp          Humidity    AirQuality    key
    ____________________    ________    __________    ___

    16-Nov-2015 00:24:22       36           81        ""
    16-Nov-2015 01:37:34       37           80        ""
    16-Nov-2015 02:50:46       36           79        ""
    16-Nov-2015 04:03:58       37           80        ""
    16-Nov-2015 05:17:09       37           81        ""
    16-Nov-2015 06:30:21       36           79        ""
    16-Nov-2015 07:43:33       37           79        ""
    16-Nov-2015 08:56:45       37           79        ""
    16-Nov-2015 10:09:57       37           85        ""
    16-Nov-2015 11:23:09       37           80        ""
```

Input Arguments

`stream` — Object connected to event stream
`KafkaStream` object | `InMemoryStream` object | `TestStream` object

Object connected to an event stream, specified as a `KafkaStream`, `InMemoryStream`, or `TestStream` object.

`tt` — Input timetable
timetable

Input timetable.

`ks` — Object connected to Kafka stream topic
`KafkaStream` object

Object connected to a Kafka stream topic, specified as a `KafkaStream` object.

`action` — Action to take if topic does not exist
`"create"` (default) | `"fail"`

Action to take if the topic to write the timetable to does not exist, specified as one of the following values:

- `"create"` — Creates new topic, if you have the required permissions on the Kafka host.
- `"fail"` — Does not create a new topic and the write operation fails.

Data Types: `char` | `string`

Version History

Introduced in R2022b

See Also

`readtimetable` | `kafkaStream` | `inMemoryStream` | `testStream`

Topics
“Process Kafka Events Using MATLAB” on page 11-5

Streaming Topics

- “Streaming Data Framework for MATLAB Production Server Basics” on page 11-2
- “Process Kafka Events Using MATLAB” on page 11-5
- “Connect to Secure Kafka Cluster” on page 11-9
- “Test Streaming Analytic Function Using Local Test Server” on page 11-12
- “Deploy Streaming Analytic Function to MATLAB Production Server” on page 11-17
- “Obtain Kafka Event Stream Log Files” on page 11-21

Streaming Data Framework for MATLAB Production Server Basics

Use Streaming Data Framework for MATLAB Production Server to read from and write to event streaming platforms, such as Kafka. Using this framework, you can:

1. Develop a streaming analytic function in MATLAB that filters, transforms, records, or processes event stream data.
2. Connect to a streaming source and test how the analytic function reads from and writes to event streams by using Streaming Data Framework for MATLAB Production Server functions.
3. Simulate the production environment for testing your streaming analytic algorithms (requires MATLAB Compiler SDK).
4. Package the analytic function (requires MATLAB Compiler SDK) and deploy it to MATLAB Production Server.

Install Streaming Data Framework for MATLAB Production Server

Install the Streaming Data Framework 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\streaming\Examples`, where `*support_package_root*` is the root folder of support packages on your system. To get the path to this folder, use this command:

```
fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','mps','streaming','Examples')
```

System Requirements

Streaming Data Framework for MATLAB Production Server has the same system requirements as MATLAB. For more information, see System Requirements for MATLAB.

Write Streaming Analytic MATLAB Function

The event stream analytic function typically consumes a stream of input events and can produce a stream of output events. It can filter, transform, record, or process the stream of events by using any MATLAB functionality that is deployable to MATLAB Production Server.

Event stream analytic functions process windows or batches of events. An event consists of three parts:

- Key — Identifies the event source
- Timestamp — Indicates the time at which the event occurred
- Body — Contains event data, specified as an unordered set of (name, value) pairs

Analytic functions read events into a timetable. Each row of the timetable represents a streaming event, typically in chronological order. If the analytic function produces results, they must also be timetables.

When processing a stream, you can call an analytic function several times, because the window size is typically much smaller than the number of messages in the stream. The stateless execution model of

MATLAB Production Server isolates the processing of each window, so the processing of one window does not affect the processing of the next. Stateful functions that require interaction between the processing of consecutive windows specify a MATLAB structure that is preserved between windows and passed to the next invocation of the analytic function.

An analytic function can have one of three signatures:

Function Signature	**Description**
results = *analyticFcn*(data)	Stateless analytic function that emits a stream of results
[results, state] = *analyticFcn*(data, state)	Stateful analytic function that preserves state between batches and emits a stream of results
analyticFcn(data)	Stateless analytic function that does not emit a stream of results

Stateless Analytic Function

The following `plotSierpinski` function is an example of a stateless analytic function. `plotSierpinski` plots the X and Y columns of the input timetable. The source code for this function and a script to run it is located in the `\Examples\ExportOptions` folder.

```
function howMany = plotSierpinski(xyData)

    hold on
    arrayfun(@(x,y)plot(x,y,'ro-', 'MarkerSize', 2), [xyData.X], [xyData.Y]);
    hold off
    drawnow
    count = height(xyData);
    howMany = timetable(xyData.Properties.RowTimes(end), count);
end
```

Stateful Analytic Function

The following `recamanSum` function is an example of a stateful analytic function. In stateful functions, the data state is shared between events and past events can influence the way current events are processed. `recamanSum` computes the cumulative sum of a numeric sequence. In returns two values:

1. `cSum` — A table that contains the cumulative sum of the elements in the stream
2. `state` — A structure that contains the final value of the sequence

The source code for the `recamanSum` function, it initialization function `initRecamanSum`, and the scripts used to run the analytic function are located in the `\Examples\Numeric` folder.

```
function [cSum, state] = recamanSum(data, state)
    timestamp = data.Properties.RowTimes;
    key = data.key;

    sum = cumsum(data.R) + state.cumsum;

    state.cumsum = sum(end);

    cSum = timetable(timestamp, key, sum);
end
```

Process Kafka Events Using MATLAB

To process events from a stream, you create an object to connect to the stream, read events from the stream, iterate the streaming analytic function to process the several windows of events, and, if the analytic function produces results, create a different stream object to write the results to stream.

The following code sample gives an overview of processing one window of events using the framework. Assume that you have a Kafka host running at the network address `kafka.host.com:9092` that has a topic `recamanSum_data`. Also, assume that the `recamanSum_data` topic contains the first 1000 elements of the Recamán sequence..

1. Create a `KafkaStream` object for reading from and writing to the `recamanSum_data` topic.

   ```
   inKS = kafkaStream("kafka.host.com",9092,"recamanSum_data");
   ```
2. Read events from the `recamanSum_data` topic into a timetable `tt`.

   ```
   tt = readtimetable(inKS);
   ```
3. Call the `recamanSum` function and calculate the cumulative sum of a part of Recamán's sequence in `tt`. Since `recamanSum` is a stateful function, first call the `initRecamSum` function, which initializes state.

   ```
   state = initRecamanSum();
   [results, state] = recamanSum(tt,state);
   ```

For a detailed example of how to process several windows of events, see “Process Kafka Events Using MATLAB” on page 11-5.

Simulate Production Using Development Version of MATLAB Production Server

Before deploying to MATLAB Production Server, you can test the streaming analytic function using the development version of MATLAB Production Server, which acts as a local test server. For a detailed example, see “Test Streaming Analytic Function Using Local Test Server” on page 11-12.

Deploy Streaming Analytic to MATLAB Production Server

You can also package the analytic function and deploy it to MATLAB Production Server. For a detailed example, see “Deploy Streaming Analytic Function to MATLAB Production Server” on page 11-17.

See Also

`readtimetable` | `writetimetable` | `kafkaStream` | `inMemoryStream` | `testStream`

Related Examples

- “Process Kafka Events Using MATLAB” on page 11-5
- “Test Streaming Analytic Function Using Local Test Server” on page 11-12
- “Deploy Streaming Analytic Function to MATLAB Production Server” on page 11-17
- “Obtain Kafka Event Stream Log Files” on page 11-21

Process Kafka Events Using MATLAB

This example shows how to use the Streaming Data Framework for MATLAB Production Server to process events from a Kafka stream. The example provides and explains the `recamanSum` and `initRecamanSum` streaming analytic functions that process event streams, and the `demoRecaman` script that creates event streams, validates event stream creation, uses the streaming analytic function to process event streams, and writes the results to an output stream.

The example functions and script are located in the *`support_package_root`*`\mps\streaming\Examples\Numeric` folder, where *`support_package_root`* is the root folder of support packages on your system. To get the path to this folder, use this command:

```
fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','mps','streaming','Examples','Numeric')
```

Prerequisites

- You must have Streaming Data Framework for MATLAB Production Server installed on your system. For more information, see “Install Streaming Data Framework for MATLAB Production Server” on page 11-2.
- You must have a running Kafka server where you have the necessary permissions to create topics. The example assumes that the network address of your Kafka host is `kafka.host.com:9092`.

Write Streaming Analytic MATLAB Function

For this example, use the sample MATLAB functions `recamanSum` and `initRecamanSum`. Later, you iterate the `recamanSum` streaming function over several events to compute results.

Write Stateful Function

The `recamanSum` function is stateful. In stateful functions, the data state is shared between events, and past events can influence the way current events are processed. `recamanSum` computes the cumulative sum of a numeric sequence in stream variable `R`, and returns a table `cSum` and structure `state`. The table `cSum` contains the cumulative sum of the elements in `R` along with timestamps. The structure `state` contains the final value of the sequence in its field `cumsum`.

```
function [cSum, state] = recamanSum(data, state)
    timestamp = data.Properties.RowTimes;
    key = data.key;

    sum = cumsum(data.R) + state.cumsum;

    state.cumsum = sum(end);

    cSum = timetable(timestamp, key, sum);
end
```

Write State Initialization Function

The `initRecamanSum` function initializes state for the first iteration of the `recamanSum` function.

```
function state = initRecamanSum(config)
    state.cumsum = 0;
end
```

Create Sample Stream Events

To run the example, you require sample streaming data. The `demoRecaman` script contains the following code to create streaming data that consists of the first 1000 elements of Recamán's sequence and also contains code to write the sequence to a Kafka topic `recamanSum_data`.

1. Set the Kafka hostname and port number.

   ```
   kafkaHost = "kafka.host.com";
   kafkaPort = 9092;
   ```
2. Create the first 1000 elements of Recamán's sequence.

   To create the sequence, you can use the following `recamanTimeTable` function also located in the `\Examples\Numeric` folder. `recamanTimeTable` creates a timetable containing the first `N` elements of Recamán's sequence.

   ```
   function tt = recamanTimeTable(N)
       rs = zeros(1,N);
       for k=2:N
           n = k-1;
           subtract = rs(k-1) - n;
           if  subtract > 0 && any(rs == subtract) == false
               rs(k) = subtract;
           else
               rs(k) = rs(k-1) + n;
           end
       end

       incr = seconds(1:N);

       thisVeryInstant = ...
           convertTo(datetime, "epochtime", "Epoch", "1970-1-1");
       thisVeryInstant = datetime(thisVeryInstant, "ConvertFrom",...
           "epochtime", "Epoch", "1970-1-1");

       thisVeryInstant.TimeZone = "UTC";
       timestamp = (thisVeryInstant - seconds(N)) + incr';

       key = (0:N-1)';
       key = string(key);
       R = rs';
       tt = timetable(timestamp,R,key);

   end
   ```
3. Store the results of `recamanTimeTable` in a timetable.

   ```
   tt0 = recamanTimeTable(1000);
   ```
4. Create a stream object connected to the `recamanSum_data` topic. Later, you write the timetable that contains the Recamán sequence to `recamanSum_data`.

   ```
   dataKS = kafkaStream(kafkaHost, kafkaPort, "recamanSum_data", Rows=100);
   ```
5. If the `recamanSum_data` topic already exists, delete it.

   ```
   try deleteTopic(dataKS); catch, end
   ```
6. Write the entire Recamán sequence to the `recamanSum_data` topic.

```
writetimetable(dataKS, tt0);
```

Validate Sample Data Creation

To validate the sample stream events that you created, confirm that the first 100 rows that you read from the `recamanSum_data` topic are the same as the sample data you created and wrote to the `recamanSum_data` topic. The `demoRecaman` script contains the following code.

1. Read one window of data (100 rows) from the `recamanSum_data` topic into a timetable `tt1`.

   ```
   tt1 = readtimetable(dataKS);
   ```
2. Check if the data read into `tt1` is equal to the first 100 elements from the Recamán sequence you wrote.

   ```
   if isequal(tt0(1:height(tt1),:), tt1)
       fprintf(1,"Success writing data to topic %s.\n", dataKS.Name);
   end
   ```
3. Stop reading from the `dataKS` stream, since later you use `dataKS` to read again from the `recamanSum_data` topic. Reading from the same topic using multiple streams is not permitted.

   ```
   stop(dataKS);
   ```

Process Stream Events with Streaming Analytic Function

Iterate the `recamanSum` streaming analytic function multiple times to read the numeric sequence from the input stream, compute its cumulative sum, and write the results to the output stream. The `demoRecaman` script contains the following code.

1. Create an output stream connected to the `recamanSum_results` topic. Use `recamanSum_results` to store the output of the `recamanSum` streaming function.

   ```
   resultKS = kafkaStream(kafkaHost,kafkaPort,"recamanSum_results", ...
       Rows=100);
   ```
2. Create an event stream processor to iterate the `recamanSum` streaming function over the input topic connected to the stream `dataKS`. Write the results to the output topic connected to the stream `resultKS`. Use a persistent storage connection named RR to store data state between iterations.

   ```
   rsp = eventStreamProcessor(dataKS,@recamanSum,@initRecamanSum,...
       StateStore="RR",OutputStream=resultKS);
   ```
3. Execute the stream function ten times. Since the window size, or the number of rows read at a time, is 100, ten iterations consumes the entire sequence of 1000 elements.

   ```
   fprintf(1,"Computing cumulative sum of Recaman sequence.\n");
   execute(rsp, 10);
   ```
4. Delete the event stream processor. This shuts down `StateStore`, which is required to run this script more than once in a row.

   ```
   clear rsp;
   ```
5. Read the results from the output stream.

   ```
   fprintf(1,"Reading results from %s.\n", resultKS.Name);
   tt2 = timetable.empty;
   for n = 1:10
   ```

```
        tt2 = [ tt2 ; readtimetable(resultKS) ];
    end

    cSum = cumsum(tt0.R);
    if tt2(end,:).sum == cSum(end)
        fprintf(1,"Cumulative sum computed successfully: %d.\n", ...
            tt2(end,:).sum);
    else
        fprintf(1,"Expected cumulative sum %d. Computed %d instead.\n", ...
            cSum(end), tt2(end,:).sum);
    end
```

When you run the entire `demoRecaman` script, you see the following output.

```
Success writing data to topic recamanSum_data.
Computing cumulative sum of Recaman sequence.
Reading results from recamanSum_results.
Cumulative sum computed successfully: 837722.
```

See Also

`readtimetable` | `writetimetable` | `kafkaStream` | `eventStreamProcessor` | `execute` | `inMemoryStream` | `testStream`

Related Examples

- “Test Streaming Analytic Function Using Local Test Server” on page 11-12
- “Deploy Streaming Analytic Function to MATLAB Production Server” on page 11-17
- “Obtain Kafka Event Stream Log Files” on page 11-21

Connect to Secure Kafka Cluster

To manage event stream processing tasks, Streaming Data Framework for MATLAB Production Server requires configuration information. For example, to connect to a secure Kafka cluster, the framework must know the security protocol and the SSL certificate to use. You provide this information by setting provider properties when creating the stream connector object. After creating the object, configuration properties are read-only. These properties are used during desktop development and then collected for deployment into production.

You can provide configuration information using two types of properties of the stream and stream processing objects:

- Named object properties — Properties required to create the MATLAB objects that interact with the stream, such as the `ConnectionTimeout` property of the `KafkaStream` object.
- Third-party provider properties — Properties that are not properties of the MATLAB objects in the streaming data framework, such as the `retention.ms` Kafka property or properties such as `security.protocol` and `ssl.truststore.type` that are required to connect to a secure Kafka cluster.

Kafka Provider Properties

When you create a `KafkaStream` object to connect to a Kafka host, specify Kafka provider properties and their corresponding values using one or more `propname,propval` input argument pairs. Use single-quotes or double-quotes around `propname`. You can specify several properties and their values in any order as `propname1,propval1`,...,`propnameN,propvalN`. For example, `kafkaStream(host,port,topic,"sasl.mechanism","SCRAM-SHA-512")` sets the Kafka property `sasl.mechanism` to `SCRAM-SHA-512`. For a complete list of Kafka properties, see Kafka Configuration in the Kafka documentation. The streaming framework provides a pass-through mechanism for these properties, where they are passed directly to the Kafka configuration mechanism without any validation.

Connect to Secure Kafka Cluster

When creating an object to connect to a secure Kafka cluster, the Kafka properties that you specify differ based on these factors:

- Whether the Kafka cluster is secured using TLS or SASL
- Whether you use the object to read from the stream or write to the stream
- Whether when using the object to read, you set the `Order` property of a `KafkaStream` object to `"EventTime"` or `"IngestTime"`.

Read Events from SSL-Secured Kafka Cluster

Specify the following Kafka properties when creating an object to read from the Kafka stream.

- `security.protocol` — Set the security protocol to `SSL`.
- `ssl.truststore.type` — Set the file format of the truststore file to `SSL` or `JKS`.
- `ssl.truststore.location` — If your server certificate is not present in your system truststore, set the location of the truststore file.

For example, the following syntax creates an object to read events from a `recamanSum_data` topic on a Kafka host located at network address `kafka.host.com:9093` in an SSL-secured cluster.

```
ks_read = kafkaStream("kafka.host.com",9093,"recamanSum_data", ...
      "security.protocol","SSL","ssl.truststore.type","PEM", ...
      "ssl.truststore.location","mps-kafka.pem")
```

Write Events to SSL-Secured Kafka Cluster

Specify the following Kafka properties when creating an object to write to the stream or to read from the stream when `Order="IngestTime"`.

- `security.protocol` — Set the security protocol to `SSL`.
- `ssl.ca.location` — Set the location of the certificate authority (CA) root certificate.

For example, the following syntax creates an object to write events to a `recamanSum_results` topic on a Kafka host located at the network address `kafka.host.com:9093` in an SSL-secured cluster.

```
outKS = kafkaStream("kafka.host.com",9093,"recamanSum_results", ...
      "security.protocol","SSL", ...
      "ssl.ca.location","my-ssl-cert.pem");
```

Read Events from SASL-Secured Kafka Cluster

To create an object to read from a SASL-secured Kafka cluster, setting the `sasl.jaas.config` Kafka property is required. The value of the `sasl.jaas.config` property is long, structured, and difficult to type. To make it easier to provide the `sasl.jaas.config` value, the framework provides two properties, `sasl.user` and `sasl.password`, that you can set instead. The framework synthesizes the value for the `sasl.jaas.config` property using the values of `sasl.user`, `sasl.password`, `security.protocol`, and `sasl.mechanism`.

Specify the following Kafka properties when creating an object to read from the stream.

- `security.protocol` — Set the security protocol to `SASL`.
- `ssl.truststore.type` — Set the file format of the truststore file to `SSL` or `JKS`.
- `ssl.truststore.location` — If your server certificate is not present in your system truststore, set the location of the truststore file.
- `sasl.mechanism` — Set the SASL mechanism used for client connections.
- `sasl.user` — Set the SASL-authorized username.
- `sasl.password` — Set the SASL password for `sasl.user`.

For example, the following syntax creates an object to read events from the `recamanSum_data` topic on a Kafka host located at the network address `kafka.host.com:9093` in a SASL-secured cluster.

```
inKS_sasl = kafkaStream("kafka.host.com",9093,"recamanSum_data", ...
      "security.protocol","SASL_SSL",
      "ssl.truststore.type","PEM",...
      "ssl.truststore.location","my-ssl-cert.pem", ...
      "sasl.mechanism","SCRAM-SHA-512", ...
      "sasl.user","sasl-consumer", ...
      "sasl.password","apachekafka")
```

Write Events to SASL-Secured Kafka Cluster

Specify the following Kafka properties when creating an object to write to the stream or to read from the stream when `Order="IngestTime"`.

- `security.protocol` — Set the security protocol to `SASL`.
- `ssl.ca.location` — Set the location of the CA root certificate.
- `sasl.mechanism` — Set the SASL mechanism used for client connections.
- `sasl.user` — Set the SASL-authorized username.
- `sasl.password` — Set the SASL password for `sasl.user`.

For example, the following syntax creates an object to write events to the `recamanSum_results` topic on a Kafka host located at the network address `kafka.host.com:9093` in a SASL-secured cluster.

```
outKS_sasl = kafkaStream("kafka.host.com",9093,"recamanSum_results", ...
        "security.protocol","SASL_SSL", ...
        "ssl.ca.location","my-ssl-cert.pem", ...
        "sasl.mechanism","SCRAM-SHA-512", ...
        "sasl.user","sasl-producer", ...
        "sasl.password","apachekafka")
```

Client-Side Authentication

To enable client-side authentication, you must set the `ssl.keystore.location` property to the location of your client certificate, the certificate the client must send to the server. If your server or client certificates are password protected, you might also need to set the `ssl.truststore.password` property and the `ssl.keystore.password` property.

See Also

`getProviderProperties` | `categoryList` | `isProperty` | `kafkaStream`

Related Examples

- “Process Kafka Events Using MATLAB” on page 11-5

External Websites

- Kafka Configuration

Test Streaming Analytic Function Using Local Test Server

This example shows how to use the development version of MATLAB Production Server to test a streaming analytic function before deployment to MATLAB Production Server. MATLAB Compiler SDK includes the development version of MATLAB Production Server, which you can use as a local test server for testing and debugging application code before deploying it to enterprise systems.

Prerequisites

- You must have Streaming Data Framework for MATLAB Production Server installed on your system. For more information, see "Install Streaming Data Framework for MATLAB Production Server" on page 11-2.
- You must have a running Kafka server where you have the necessary permissions to create topics. The example assumes that the network address of your Kafka host is `kafka.host.com:9092`.

Write Streaming Analytic MATLAB Function

For testing purposes, use the sample MATLAB functions `recamanSum` and `initRecamanSum` located in the *`support_package_root`*`\mps\streaming\Examples\Numeric` folder, where *`support_package_root`* is the root folder of support packages on your system. To get the path to this folder, use this command:

```
fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','mps','streaming','Examples','Numeric')
```

Later, you test the `recamanSum` deployable archive using the local test server.

For details about the `recamanSum` and `initRecamanSum` functions and to access the code, see "Write Stateful Function" on page 11-5 and "Write State Initialization Function" on page 11-5.

Create Sample Streaming Data

Prepare for testing by creating sample data and writing the data to a Kafka stream. For this example, you create a 1000-element Recamán sequence and write it to a Kafka topic `recamanSum_data`.

To create the Recamán sequence, you can use the following `recamanTimeTable` function, which is also located in the `\Examples\Numeric` folder. `recamanTimeTable` creates a timetable containing the first `N` elements of a Recamán sequence.

```
function tt = recamanTimeTable(N)
    rs = zeros(1,N);
    for k=2:N
        n = k-1;
        subtract = rs(k-1) - n;
        if  subtract > 0 && any(rs == subtract) == false
            rs(k) = subtract;
        else
            rs(k) = rs(k-1) + n;
        end
    end

    incr = seconds(1:N);

    thisVeryInstant = ...
```

```
        convertTo(datetime, "epochtime", "Epoch", "1970-1-1");
    thisVeryInstant = datetime(thisVeryInstant, "ConvertFrom",...
        "epochtime", "Epoch", "1970-1-1");

    thisVeryInstant.TimeZone = "UTC";
    timestamp = (thisVeryInstant - seconds(N)) + incr';

    key = (0:N-1)';
    key = string(key);
    R = rs';
    tt = timetable(timestamp,R,key);

end
```

You can use the following code to create a 1000-element Recamán sequence using the `recamanTimeTable` function and write it to the `recamanSum_data` Kafka topic. The example assumes that you have Kafka host running at the network address `kafka.host.com:9092` and you have the necessary permissions to create topics in the Kafka cluster.

```
kafkaHost = "kafka.host.com";
kafkaPort = 9092;

tt0 = recamanTimeTable(1000);

dataKS = kafkaStream(kafkaHost, kafkaPort, "recamanSum_data", Rows=100);

try deleteTopic(dataKS); catch, end

writetimetable(dataKS, tt0);

tt1 = readtimetable(dataKS);

if isequal(tt0(1:height(tt1),:), tt1)
    fprintf(1,"Success writing data to topic %s.\n", dataKS.Name);
end

stop(dataKS);
```

Simulate Production Using Local Test Server

To simulate streaming data processing in a production environment, you can run the `recamanSum` deployable archive using the development version of MATLAB Production Server and process data from the `recamanSum_data` topic.

1. Create a `KafkaStream` object connected to the `recamanSum_data` topic.

   ```
   ks = kafkaStream("kafka.host.com",9092,"recamanSum_data");
   ```
2. Create another `KafkaStream` object to write the results of the `recamanSum` analytic function to a different topic called `recaman_results`.

   ```
   outKS = kafkaStream("kafka.host.com",9092,"recamanSum_results");
   ```
3. Create an `EventStreamProcessor` object that runs the `recamanSum` function and initializes persistent state with the `initRecamanSum` function.

   Providing a persistent data storage connection name as an input argument is optional. If you do not provide one, `EventStreamProcessor` creates a connection with a unique name to cache the data state between iterations.

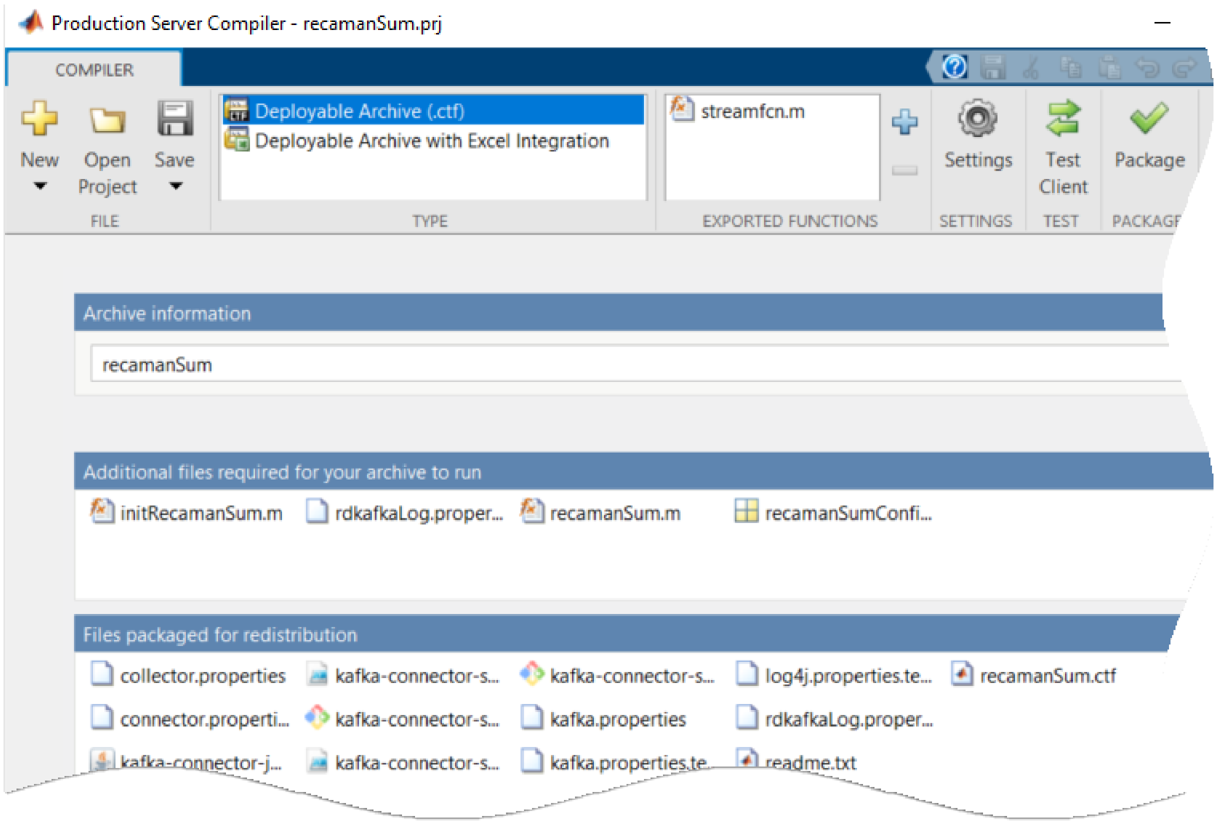
```
esp = eventStreamProcessor(ks,@recamanSum,@initRecamanSum,OutputStream=outKS);
```

4 Using the MATLAB editor, you can set breakpoints in the `recamanSum` function to examine the incoming streaming data when you start the server.

5 Start the test server.

```
startServer(esp);
```

Doing so opens the **Production Server Compiler** app with values for the streaming function `recamanSum`, the entry point function `streamfcn`, and the deployable archive `recamanSum`.


Production Server Compiler - recamanSum.prj
COMPILER
New
Open Project
Save
FILE
Deployable Archive (.ctf)
Deployable Archive with Excel Integration
TYPE
streamfcn.m
EXPORTED FUNCTIONS
Settings
SETTINGS
Test Client
TEST
Package
PACKAGE

Archive information

recamanSum

Additional files required for your archive to run

initRecamanSum.m rdkafkaLog.proper... recamanSum.m recamanSumConfi...

Files packaged for redistribution

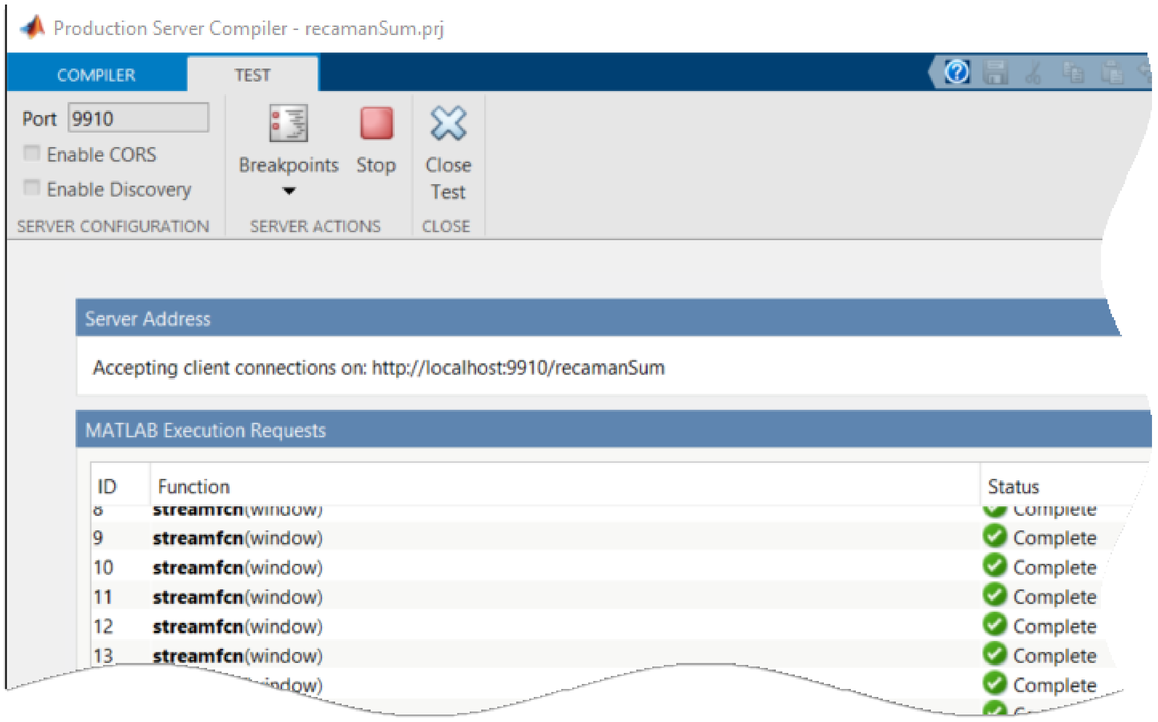
collector.properties kafka-connector-s... kafka-connector-s... log4j.properties.te... recamanSum.ctf
connector.properti... kafka-connector-s... kafka.properties rdkafkaLog.proper...
kafka-connector-j... kafka-connector-s... kafka.properties.te... readme.txt

6 Start the test server from the app by clicking **Test Client**, and then **Start**.

7 Navigate back to the MATLAB command prompt to start processing events.

```
start(esp);
```

In the **Production Server Compiler** app, you can see that the test server receives data.



Accepting client connections on: http://localhost:9910/recamanSum

ID	Function	Status
8	**streamfcn**(window)	Complete
9	**streamfcn**(window)	Complete
10	**streamfcn**(window)	Complete
11	**streamfcn**(window)	Complete
12	**streamfcn**(window)	Complete
13	**streamfcn**(window)	Complete
	indow)	Complete

8 From the MATLAB editor, if you had set breakpoints, you can use the debugger to examine the data, state, and results of the function processing. Click **Continue** to continue debugging or **Stop** when you have finished debugging.

9 From the MATLAB command prompt, stop event processing and shut down the server.

```
stop(esp);
stopServer(esp);
```

10 Read the processed results from the output stream.

```
results = readtimetable(outKS);
```

```
results =

  50×2 timetable

         timestamp            key      sum 
    ____________________    ____    ____

    20-Jan-1970 04:55:08    "0"        0
    20-Jan-1970 04:55:08    "1"        1
    20-Jan-1970 04:55:08    "2"        4
    20-Jan-1970 04:55:08    "3"       10
    20-Jan-1970 04:55:08    "4"       12

             :                :       :

    20-Jan-1970 04:55:08    "45"    1697
    20-Jan-1970 04:55:08    "46"    1732
    20-Jan-1970 04:55:08    "47"    1814
    20-Jan-1970 04:55:08    "48"    1848
    20-Jan-1970 04:55:08    "49"    1931

    Display all 50 rows.
```

See Also
`kafkaStream` | `eventStreamProcessor` | `execute` | `package` | `seek` | `start` | `startServer` | `stop` | `stopServer`

Related Examples
- "Deploy Streaming Analytic Function to MATLAB Production Server" on page 11-17

Deploy Streaming Analytic Function to MATLAB Production Server

You can package streaming analytic functions developed using Streaming Data Framework for MATLAB Production Server and deploy the packaged archive (CTF file) to MATLAB Production Server. The deployed archive expects to receive streaming data. The Kafka Connector executable pulls data from a Kafka host and pushes it to the deployed streaming archive. In the MATLAB desktop, Streaming Data Framework for MATLAB Production Server manages the Kafka connector. On a server instance, you must manage starting and stopping the Kafka .

The topic describes the Kafka connector and provides an example to process streaming data using a stateful streaming analytic function deployed to the server.

Kafka Connector Specifications

The Kafka connector is a Java program that requires at least Java 8. To use the Kafka connector, the `JAVA_HOME` environment variable on your server machine must be set to the path of your Java 8 installation.

Each deployed archive that contains a streaming analytic function requires its own Kafka connector. For example, if you have two archives, you require two connectors. You do not have to install the Kafka connector twice, but you must run it twice and have exactly one Kafka connector configuration file per archive.

The life cycle management of the Kafka connector depends on your production environment. Streaming Data Framework for MATLAB Production Server provides tools to make starting, stopping, and controlling the Kafka connector easier.

Prerequisites for Running Example

The following example provides a sample stateful streaming analytic function, shows how to package and deploy it to MATLAB Production Server, and shows how to manage the Kafka connector on the server.

To run the example, you require sample streaming data and a running MATLAB Production Server instance with a running persistence service.

Create Sample Streaming Data

Create sample streaming data and write the data to a Kafka stream. For this example, you create a 1000-element Recamán sequence and write it to a Kafka topic `recamanSum_data`. For details about creating the streaming data, see “Create Sample Streaming Data” on page 11-12.

Create Server Instance

Create a MATLAB Production Server instance to host the streaming deployable archive. For details about creating a server instance using the command line, see “Set Up MATLAB Production Server Using the Command Line”. For details about creating a server instance using the dashboard, see “Create Server Instance Using Dashboard”.

Start Persistence Service

Create a persistence service on the server instance and name the persistence connection RR. Start the persistence service. Later, when you package the streaming function into a deployable archive, you use the RR connection name. For details about creating and starting a persistence service, see “Data Caching Basics” on page 6-2.

Start Server Instance

Start the server instance that you created. For details about starting a server instance using the command line, see “Start Server Instance Using Command Line”. For details about starting a server instance using the dashboard, see “Start Server Instance Using Dashboard”.

Write Streaming Analytic MATLAB Function

For this example, use the sample MATLAB functions `recamanSum` and `initRecamanSum`, which are located in the `*support_package_root*\toolbox\mps\streaming\Examples\Numeric` folder, where `*support_package_root*` is the root folder of support packages on your system. To get the path to this folder, use this command:

```
fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','mps','streaming','Examples','Numeric')
```

Later, you package the `recamanSum` function and deploy it to MATLAB Production Server. For details about the `recamanSum` and `initRecamanSum` functions and to access the code, see “Write Stateful Function” on page 11-5 and “Write State Initialization Function” on page 11-5.

Package Streaming Analytic Function

To package the `recamanSum` streaming analytic into a deployable archive, you can run the following script. The script creates an input `KafkaStream` object `dataKS` connected to the `recamanSum_data` topic and an output `KafkaStream` object `resultKS` connected to the `recamanSum_results` topic. Then, the script uses the `streamingDataCompiler` function to launch the **Production Server Compiler** app. Using the app, you create a deployable archive `recamanSum.ctf` suitable for deployment to MATLAB Production Server. Provide the `StateStore` input argument in the call to `streamingDatacompiler` and set its value to RR. RR is the persistence connection name you created in “Start Persistence Service” on page 11-18.

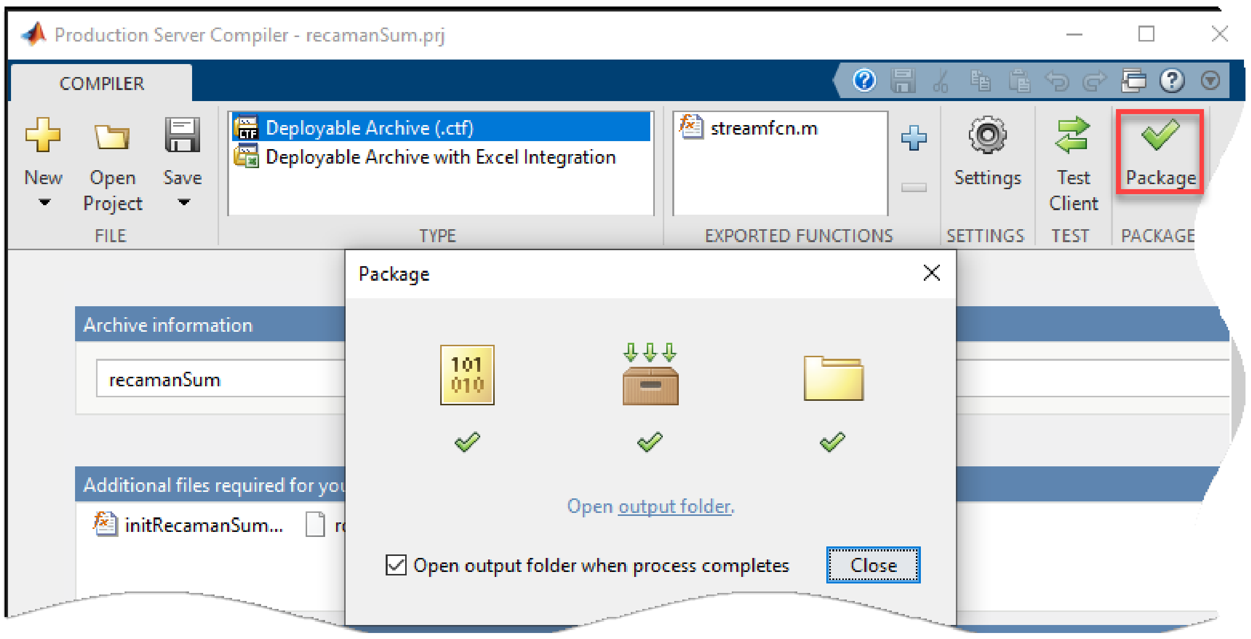
```
kafkaHost = "kafka.host.com";
kafkaPort = 9092;

dataKS = kafkaStream(kafkaHost, kafkaPort, "recamanSum_data", Rows=100);

resultKS = kafkaStream(kafkaHost, kafkaPort, "recamanSum_results", ...
    Rows=100);

archive = streamingDataCompiler("recamanSum", dataKS, resultKS, ...
    InitialState="initRecamanSum", StateStore="RR");
```

From the **Production Server Compiler** app, click **Package** to create the `recamanSum` archive. When the packaging process finishes, open the output folder. In the output folder, navigate to the `for_distribution` folder. The `for_distribution` folder contains the `recamanSum.ctf` deployable archive and Kafka connector scripts that you use later.



Deploy Streaming Analytic Function to Server

Deploy the `recamanSum` archive to a running MATLAB Production Server instance. If you manage the server using the command line, copy the `recamanSum` archive to the `auto_deploy` folder of your server instance. For other ways to deploy, see “Deploy Archive to MATLAB Production Server”.

Start Kafka Connector

Depending on the operating system of your server instance, enter the following commands at the system prompt to start the Kafka connector script `kafka-connector-start`. The Kafka connector pulls data from the Kafka host and pushes it to the deployed streaming archive.

The output of the start script is a process ID (PID). Save the value of the PID. You use this ID to stop the Kafka Connector process later.

Windows

```
powershell -executionPolicy bypass -File kafka-connector-start.ps1 -out out.log -err error.log `
-c collector.properties -k kafka.properties
```

Linux

```
 chmod +x kafka-connector-start.sh
```

```
./kafka-connector-start.sh -out out.log -err error.log -c collector.properties -k kafka.properties
```

Read Processed Data From Output Stream

After you start the Kafka Connector, the server starts receiving several requests. The deployed `recamanSum` archive receives streaming data from the input Kafka stream as input and calculates the cumulative sum of the Recamán sequence. Wait a few seconds for the server to finish processing these requests.

Create another `KafkaStream` object to read the result from the output topic.

```
readStream = kafkaStream("kafka.host.com", 9092, "recamanSum_results");
```

Call `readtimetable` to read the output data.

```
result = readtimetable(readStream)
```

Stop Kafka Connector

Depending on the operating system of your server instance, enter the following commands at the system prompt to stop the Kafka Connector script `kafka-connector-stop`. Replace *`PID`* with the process ID that you save when you start the connector.

Windows

```
powershell -executionPolicy bypass -File kafka-connector-stop.ps1 PID
```

Linux

```
 chmod +x kafka-connector-stop.sh
 ./kafka-connector-stop.sh PID
```

See Also

`streamingDataCompiler` | `package`

Related Examples

- “Test Streaming Analytic Function Using Local Test Server” on page 11-12

Obtain Kafka Event Stream Log Files

When processing Kafka stream events using a `KafkaStream` object, use log files to help debug event streaming issues. These files contain all warnings, errors, and other information related to reading and writing events to and from Kafka streams. You can generate log files from these sources:

- `KafkaStream` objects connected to the Kafka topic — Each object generates a log file containing information about reads from the Kafka server.
- `librdkafka` Kafka C/C++ client library — This library generates a log file containing information about writes from the Kafka server.

For both sources, you can configure the log level, which controls the amount of output written to the log files. You can set these log levels, listed in order from least to most verbose:

1. `off` — Logging not enabled (default for `librdkafka` library)
2. `fatal` — Log only errors that force the Kafka connection to shut down
3. `error` — Log all errors (default for `kafkaStream` objects)
4. `warn` — Log all errors and warnings
5. `info` — Log errors, warnings, and high-level information about major streaming activities
6. `debug` — Log debugging information in addition to information in previously described options
7. `trace` — Log stack trace information in addition to information in previously described options

You can also configure the log level in your deployed applications.

Configure kafkaStream Object Logging

The log file for each `KafkaStream` object is generated to this file in your current folder.

log\*topic_name*.log

topic_name is the name of the Kafka topic that the `KafkaStream` object is connected to. If the `log` folder does not exist, the `KafkaStream` object creates it.

By default, the log level for these objects is set to `error`. To change the log level, update the `log4j.properties.template` file contained in your installation of Streaming Data Framework for MATLAB Production Server. Open the file by entering this command at the MATLAB command prompt.

```
open(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','mps','streaming', ...
                 '+matlab','+io','+stream','+event','+kafka','collector','log4j.properties.template'))
```

To change the log level in the `log4j.properties.template` file, you must update the log level in all locations where the log level is specified. For example, to change the log level from `error` to `debug`, in the `log4j.properties.template` file, search for the lines containing this code.

```
.level = error
```

Within each line, update this code to the following:

```
.level = debug
```

Alternatively, to quickly review information only about the last error received from the Kafka topic, use the `loggederror` function. Pass your `KafkaStream` object as an input argument to this function.

To generate the logs, the streaming data framework uses version 2 of the Log4j Java library. For more details on this library, see `https://logging.apache.org/log4j/2.x/`.

Configure librdkafka Library Logging

When logging for the `librdkafka` library is enabled, the library generates a single file in the current folder named `mw_rdkafka.log`. If the log file does not already exist, the `KafkaStream` object creates it.

By default, the log level for the `librdkafka` library is set to `off`. To enabled logging, use this MATLAB command:

```
matlab.io.stream.event.kafka.internal.admin(log=logLevel);
```

`logLevel` is a string or character vector indicating the log level described earlier, such as `"error"`, `"warn"` or `"debug"`. For example, setting `logLevel` to `"debug"` enables logging and sets the log level to debug mode.

Configure Logging in Deployed Applications

When you package a deployed application using a project file created by the `package` or `streamingDataCompiler` functions, the packaging process copies configuration files into the `for_redistribution` folder of your application. You use these files when you start the Kafka Connector to push streaming data from your Kafka topic to your application hosted by MATLAB Production Server. To enable logging in the Kafka Connector, copy the `log4j.properties.template` file to a new file named `log4j.properties`. Then, edit this new file to choose the log level and set the name of the log file.

To set the log level, set the `*.level` properties to the desired log level, as described in the "Configure kafkaStream Object Logging" on page 11-21 section. To set the name of the log file, change the text `!ArchiveName!` to the name of the desired log file. The `!ArchiveName!` text appears on these lines:

```
appender.rolling.fileName = ${basePath}!ArchiveName!.out
appender.rolling.filePattern = ${basePath}!ArchiveName!%i.out
```

For example, to send output log messages to the file `RecamanSum.out`, change these lines to:

```
appender.rolling.fileName = ${basePath}RecamanSum.out
appender.rolling.filePattern = ${basePath}RecamanSum%i.out
```

To control `librdkafka` logging, edit the `rdkafkalog.properties` file, which is located at this path:

```
open(fullfile(matlabshared.supportpkg.getSupportPackageRoot,'toolbox','mps','streaming', ...
              '+matlab','+io','+stream','+event','+kafka','collector','rdkafkalog.properties'))
```

This file is installed with read-only access, so you must make the file writeable before editing it. For your changes to take effect, you must edit this file before using the `package` or `streamingDataCompiler` function to create the deployable archive. Packaging incorporates the `rdkafka.properties` file into your application. Although you cannot change this file after packaging, you can use an environment variable to control the location from which MATLAB Production Server loads the `rdkafka.properties` file.

You can change the log level and log file name by setting values in `rdkafka.properties`. For example, to change the log level from `fatal` (default) to `info`, change the `level=fatal` line to `level=info`.

Similarly, the `file` property sets the full path to the output log file. The default is `mw_rdkafka.log`, as in:

```
file=mw_rdkafka.log
```

With no folder specification, this file is in the top-level folder of the MATLAB Production Server instance that hosts the deployed streaming application, which is the parent folder of the `auto_deploy` folder. If you deploy multiple streaming applications to the same MATLAB Production Server instance, make this filename application-specific. For example:

```
file=recamanSum_rdkafka.log
```

If you leave the filename unchanged, this file is shared by all streaming applications hosted by that server.

To change the folder where the file appears, specify a full or relative path using platform-appropriate syntax.

Windows	Linux or Mac
`file=c:\mps_logs \recamanSum_rdkafka.log`	`file=/myusername/mps_logs/ recamanSum_rdkafka.log`

To temporarily change the location from which MATLAB Production Server loads the `rdkafka.properties` file, use the environment variable `MW_IOSTREAM_RDKAFKA_LOG_CONFIG`. You must set this environment variable before you start the MATLAB Production Server instance hosting your application. You must also set this variable in the environment from which you start that instance. Otherwise, MATLAB Production Server is unable to locate the environment variable. This environment variable setting changes the location of `rdkafka.properties` for all applications hosted by that instance of MATLAB Production Server. Use this environment variable for temporary troubleshooting only.

To temporarily cause all streaming applications to configure `rdkafka` logging according to the file and log level set in a nondeployed `rdkafka.properties` file, set `MW_IOSTREAM_RDKAFKA_LOG_CONFIG` to the full path of the nondeployed file. For example:

Windows

```
set MW_IOSTREAM_RDKAFKA_LOG_CONFIG=c:\mps_logs\recamanSum_rdkafka.log
```

Linux or Mac

```
setenv MW_IOSTREAM_RDKAFKA_LOG_CONFIG /myusername/mps_logs/recamanSum_rdkafka.log
```

Provide Log Files to MathWorks Technical Support

For additional help debugging, you can provide your log files to MathWorks Technical Support. Before providing the log files, set the log level to `trace`.

1. Locate the log files, using the previously provided information.
2. Attach the log files to a new or existing ticket for MathWorks Technical Support. See Contact Support.

See Also
`loggederror`