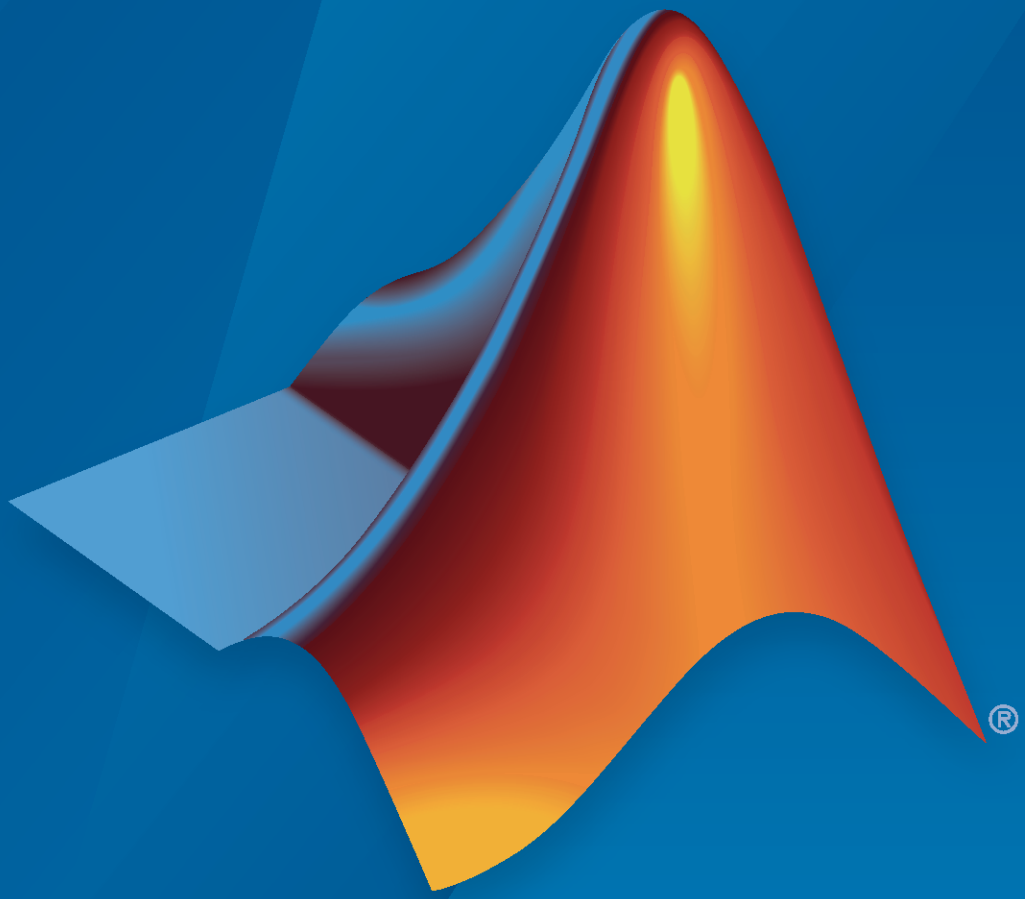
MATLAB® Compiler SDK™

MATLAB® Production Server™ Testing Guide



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® Compiler SDK™ MATLAB® Production Server™ Testing Guide

Revision History

March 2015	Online only	New for Version 6.0 (Release R2015a)
September 2015	Online only	Revised for Version 6.1 (Release 2015b)
March 2016	Online only	Revised for Version 6.2 (Release 2016a)
September 2016	Online only	Revised for Version 6.3 (Release R2016b)
March 2017	Online only	Revised for Version 6.3.1 (Release R2017a)
September 2017	Online only	Revised for Version 6.4 (Release R2017b)
March 2018	Online only	Revised for Version 6.5 (Release R2018a)
September 2018	Online only	Revised for Version 6.6 (Release R2018b)
March 2019	Online only	Revised for Version 6.6.1 (Release R2019a)
September 2019	Online only	Revised for Version 6.7 (Release R2019b)
March 2020	Online only	Revised for Version 6.8 (Release R2020a)
September 2020	Online only	Revised for Version 6.9 (Release R2020b)
March 2021	Online only	Revised for Version 6.10 (Release R2021a)
September 2021	Online only	Revised for Version 6.11 (Release R2021b)
March 2022	Online only	Revised for Version 7.0 (Release R2022a)
September 2022	Online only	Revised for Version 7.1 (Release R2022b)
March 2023	Online only	Revised for Version 7.2 (Release R2023a)

Contents

1 Deployable Archive Creation

2 Write Deployable MATLAB Code

3 Persistence

4 MATLAB Production Server Integration Testing

1

Deployable Archive Creation

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” on page 1-5.

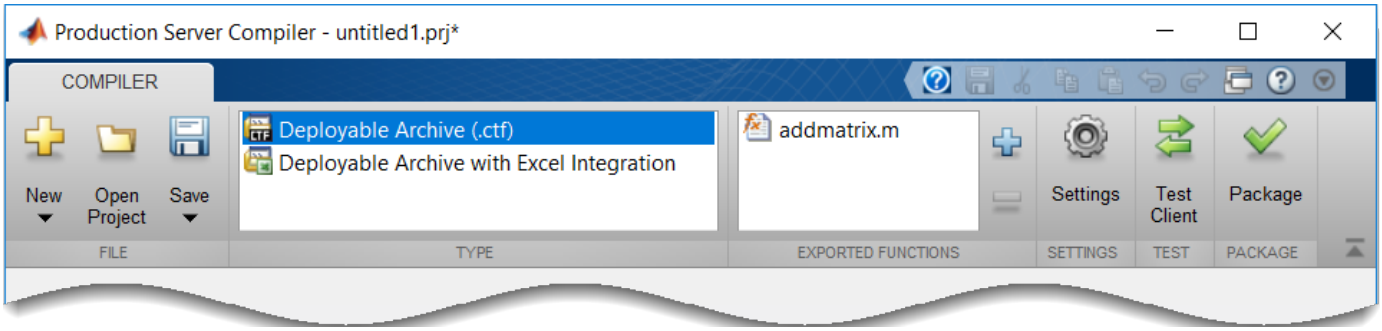
1. To open the **Production Server Compiler** app, type `productionServerCompiler` at the MATLAB prompt.

   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.



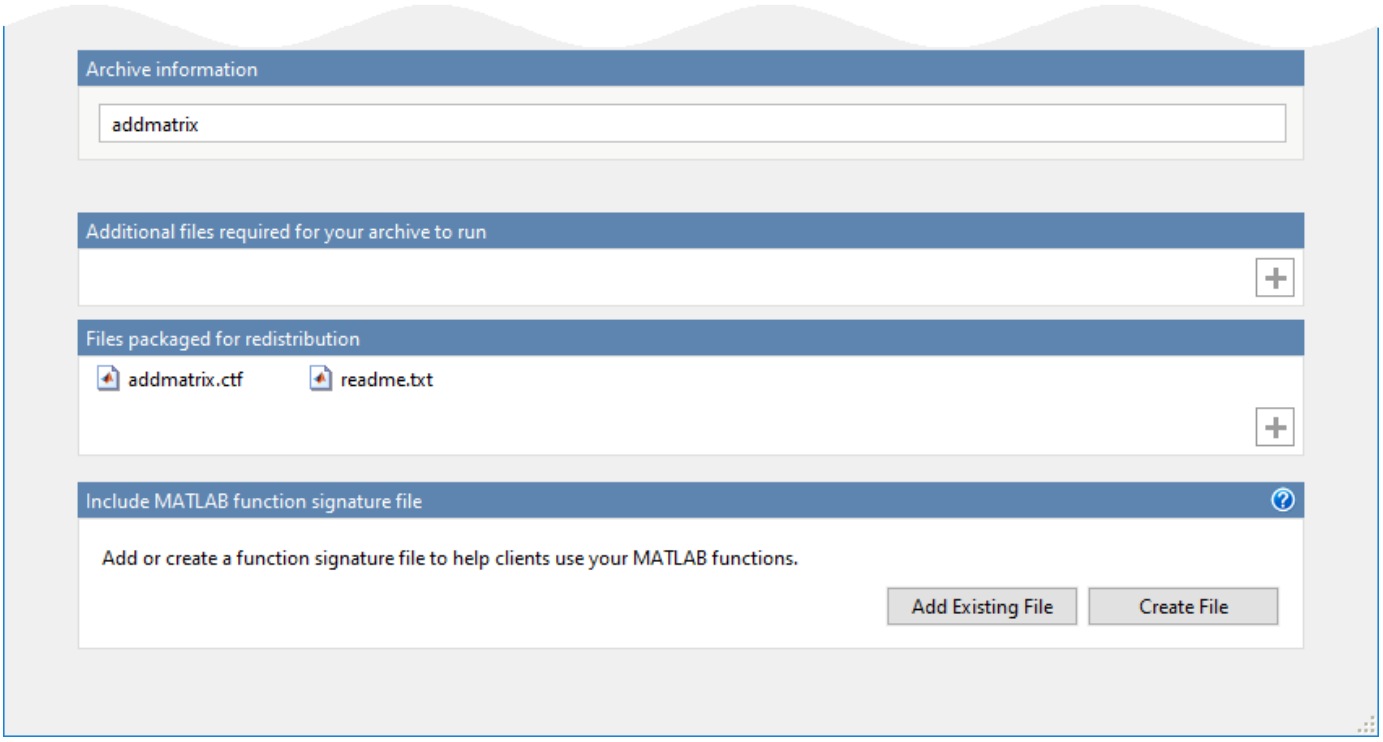
Customize Application and Its Appearance

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”.
- **Files packaged for redistribution** — Files that are installed with your archive. These files include:

  - Generated deployable archive
  - Generated `readme.txt`

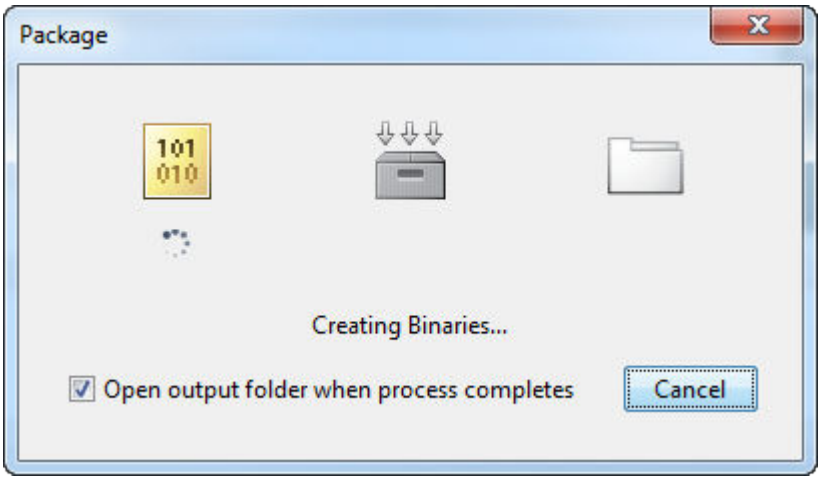
  See “Specify Files to Install with Application”.
- **Include MATLAB function signature file** — Add or create a function signature file to help clients use your MATLAB functions. See “MATLAB Function Signatures in JSON”.



Package Application

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

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



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

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

- `for_redistribution` — Folder containing the archive *`archiveName`*`.ctf`
- `for_testing` — Folder containing the raw generated files to create the installer

- `PackagingLog.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.
  - `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” on page 4-3
- Production Server Compiler
- “Deploy Archive to MATLAB Production Server” (MATLAB Production Server)
- “MATLAB Function Signatures in JSON”
- “JSON Representation of MATLAB Data Types” (MATLAB Production Server)

Create and Install a Deployable Archive with Excel Integration for MATLAB Production Server

Supported Platform: Microsoft® Windows only.

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

Prerequisites

MATLAB Compiler SDK requires .NET framework 4.0 or later to build Excel add-ins for MATLAB Production Server.

To generate the Excel add-in file (.xla), 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.

Create Function in MATLAB

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

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

```
function y = mymagic(x)

y = magic(x);
```

At the MATLAB command prompt, enter `mymagic(3)`.

The output is:

```
ans =
     8     1     6
     3     5     7
     4     9     2
```

Create Deployable Archive with Excel Integration Using Production Server Compiler App

Package the function into a deployable archive with Excel integration 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 with Excel Integration Using compiler.build.excelClientForProductionServer” on page 1-10.

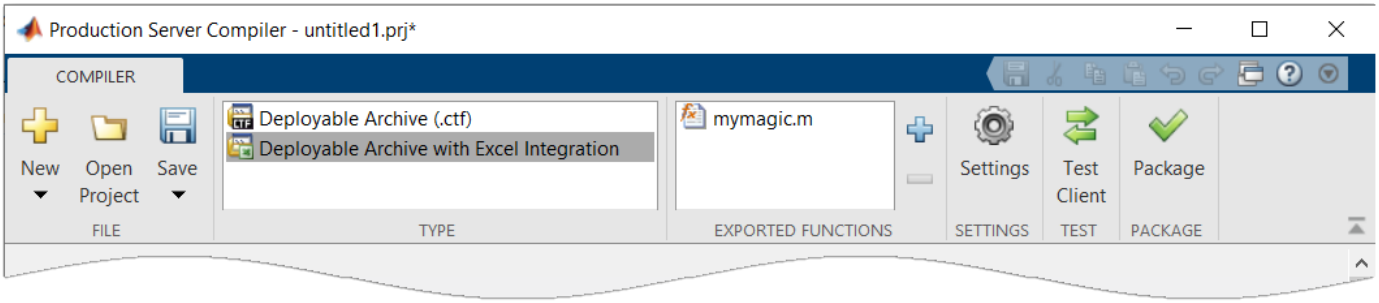
1. To open the **Production Server Compiler** app, type `productionServerCompiler` at the MATLAB prompt.

   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 with Excel integration**.
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 `mymagic.m` to the list of main files.



Customize the Application and Its Appearance

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

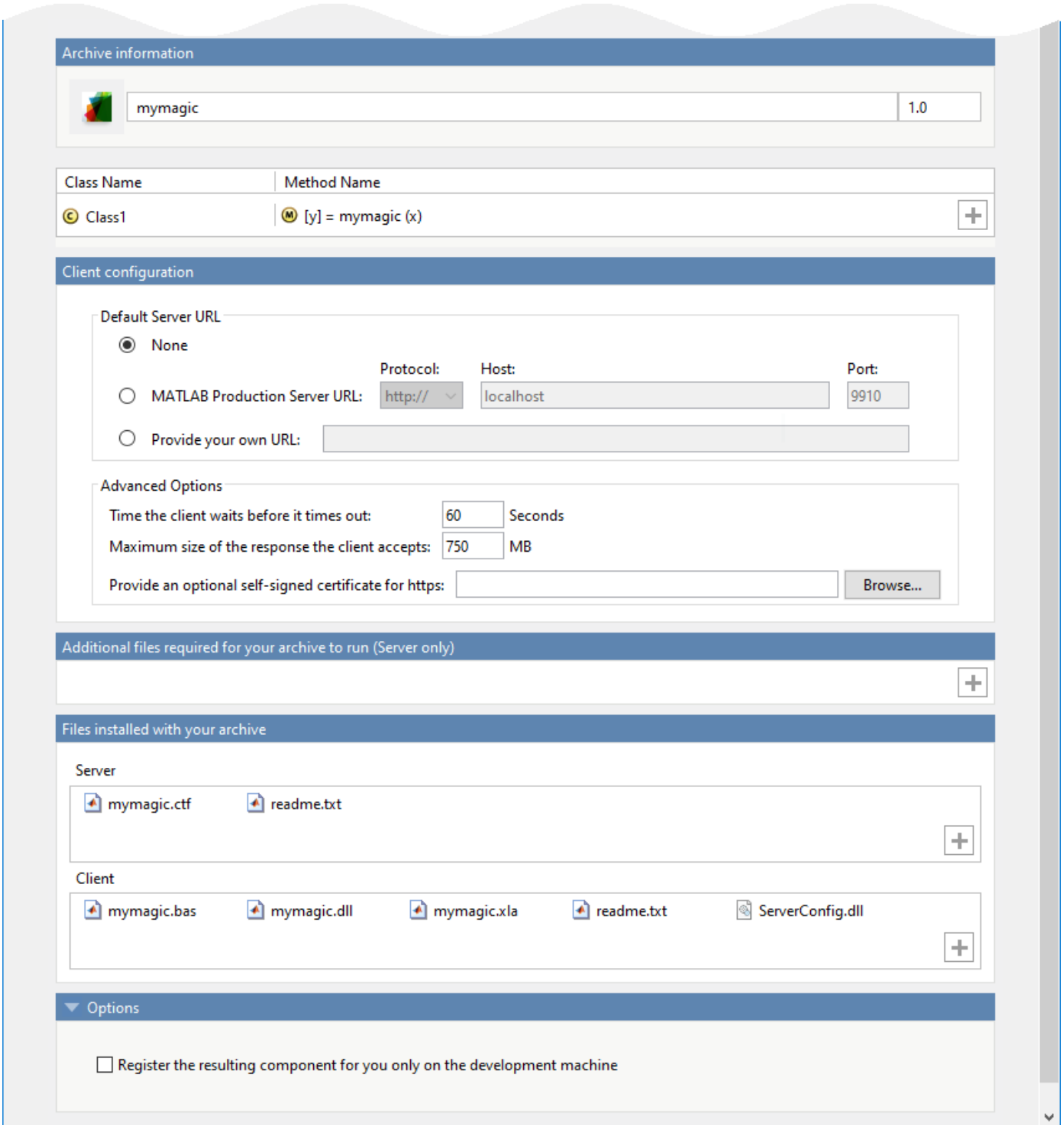
- **Archive information** — Editable information about the deployed archive with Excel integration.
- **Client configuration** — Configure the MATLAB Production Server client. Select the **Default Server URL**, decide wait time-out, and maximum size of response for the client, and provide an optional self-signed certificate for `https`.
- **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”.
- **Files installed with your archive** — Files that are installed with your archive on the client and server. The files installed on the server include:

  - Generated deployable archive (CTF file)
  - Generated `readme.txt`

  The files installed on the client include:

  - `mymagic.bas`
  - `mymagic.dll`
  - `mymagic.xla`
  - `readme.txt`
  - `ServerConfig.dll`

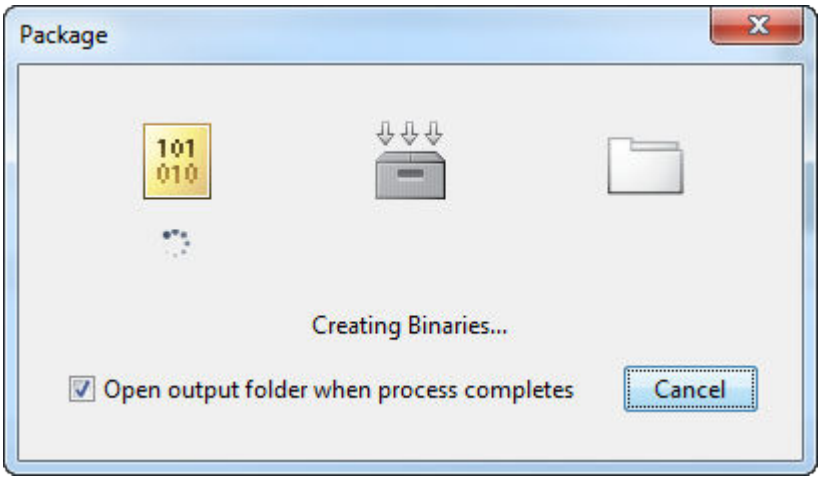
  See “Specify Files to Install with Application”.
- **Options** — The option **Register the resulting component for you only on the development machine** exclusively registers the packaged component for one user on the development machine.



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 **Open output folder when process completes** is selected.

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

- `for_redistribution` — Folder containing the installer to distribute the archive on the MATLAB Production Server client and server
- `for_redistribution_files_only` — Folder containing the files required for redistributing the application on the MATLAB Production Server client and server
- `for_testing` — Folder containing the raw generated files to create the installer
- `PackagingLog.html` — Log file generated by MATLAB Compiler SDK

Create Deployable Archive with Excel Integration Using compiler.build.excelClientForProductionServer

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

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

```
buildResults = compiler.build.productionServerArchive('mymagic.m');
```

2 Build the deployable archive with Excel integration using the `compiler.build.excelClientForProductionServer` function.

```
mpsxlResults = compiler.build.excelClientForProductionServer(buildResults, ...
'Verbose','on');
```

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

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 `mymagicexcelClientForProductionServer` in your current working directory:

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

- `mymagic.bas` — VBA module file that can be imported into a VBA project.
- `mymagic.dll` — Dynamic library required by the Excel add-in.
- `mymagic.reg` — Text file that contains information on unresolved symbols.
- `mymagic.xla` — Excel add-in that can be installed directly in Excel.
- `mymagicClass.cs` — Text file that contains information on unresolved symbols.
- `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.
- `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.

Note The generated Excel add-in does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

Install the Deployable Archive with Excel Integration

You must deploy the archive to a MATLAB Production Server instance before you can use the add-in in Excel.

To install the deployable archive on a server instance:

1. Locate the archive in the `for_redistribution_files_only\server\` folder if you used the Production Server Compiler, or the `addmatrixproductionServerArchive` folder if you used the `compiler.build.productionServerArchive` function.

   For this example, the file name is `mymagic.ctf`.
2. Copy the archive file to the `auto_deploy` folder of the server instance. The server instance automatically deploys it and makes it available to interested clients.

For more information, see “MATLAB Production Server” documentation.

See Also

Production Server Compiler | `mcc`

Create Microservice Docker Image

Supported platform: Linux, Windows, macOS

This example shows how to create a microservice Docker image. The microservice image created by MATLAB Compiler SDK provides an HTTP/HTTPS endpoint to access MATLAB code.

You package a MATLAB function into a deployable archive, and then create a Docker image that contains the archive and a minimal MATLAB Runtime package. You can then run the image in Docker and make calls to the service using any programming language that has HTTP libraries, including MATLAB Production Server client APIs.

This option is best for developers who want to incorporate a MATLAB algorithm or Simulink simulation within a larger application as a service, or to provide a synchronous request-response backend API service. To create a Docker image that contains a standalone application, see “Package MATLAB Standalone Applications into Docker Images”.

Prerequisites

- Verify that you have MATLAB Compiler SDK installed on the development machine.
- Verify that you have Docker installed and configured on the development machine by typing `[~,msg] = system('docker version')` in a MATLAB command window.

  Note If you are using WSL, use `[~,msg] = system('wsl docker version')` instead.

  If you do not have Docker installed, follow the instructions on the Docker website to install and set up Docker.

  `docs.docker.com/engine/install/`

  To build microservice images on Windows, you must install either Docker Desktop or Docker on Windows Subsystem for Linux v2 (WSL2).

  - To install Docker Desktop, see `docs.docker.com/desktop/windows/install/`.
  - To install Docker on WSL2, see `https://www.mathworks.com/matlabcentral/answers/1758410-how-do-i-install-docker-on-wsl2`.
- If the computer you are using is not connected to the Internet, you must download the MATLAB Runtime installer for Linux from a computer that is connected to the Internet and transfer the installer to the offline computer. Then, run the command `compiler.runtime.createInstallerDockerImage(filepath)`, where `filepath` is the path to the transferred MATLAB Runtime installer archive.

  You can download the installer from the MathWorks website.

  `https://www.mathworks.com/products/compiler/matlab-runtime.html`

Create MATLAB Function

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

For this example, write a function named `mymagic.m` using the following code.

```
function y = mymagic(x)
y = magic(x);
```

At the MATLAB command prompt, enter `mymagic(5)`.

The output is a 5-by-5 magic square matrix.

```
ans =

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

Create Deployable Archive

Package the `mymagic` function into a deployable archive using the `compiler.build.productionServerArchive` function.

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

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

```
mpsResults = compiler.build.productionServerArchive('mymagic.m',...
'FunctionSignatures','mymagicFunctionSignatures.json',...
'ArchiveName','magicarchive','Verbose','on')
```

```
mpsResults =

  Results with properties:

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

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

Once the build is complete, the function creates a folder named `magicarchiveproductionServerArchive` in your current directory that contains the deployable archive.

Package Deployable Archive into Docker Image

Build the microservice Docker image using the `mpsResults` object that you created.

Specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.package.microserviceDockerImage`.

```
compiler.package.microserviceDockerImage(mpsResults,'ImageName','micro-magic')
```

The function generates the following files within a folder named `micro-magicmicroserviceDockerImage`:

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

Test Docker Image

Note If Docker is running in a WSL2 session, preface the following commands with `wsl`.

1. In a Linux terminal, verify that your `micro-magic` image is in your list of Docker images.

   ```
   docker images
   ```

   ```
   REPOSITORY                                          TAG           IMAGE ID            CREATED
   micro-magic                                         latest        4401fa2bc057        23 seconds
   matlabruntime/r2023a/update0/4200000000000000      latest        5259656e4a32        24 hours ag
   ```

2. Run the `micro-magic` microservice image in Docker.

   ```
   docker run --rm -p 9900:9910 micro-magic
   ```

   Port 9910 is the default port exposed by the microservice within the Docker container. You can map it to any available port on your host machine. For this example, it is mapped to port 9900.

   You can specify additional options in the Docker command. For a complete list of options, see “Microservice Command Arguments” on page 1-17.

3. Once the container is running in Docker, you can check the status of the service by opening the following URL in a web browser:

   `http://`*`hostname`*`:9900/api/health`

   Note Use `localhost` as the hostname if Docker is running on the same machine as the browser.

   If the service is ready to receive requests, you see the following message:

   ```
   "status:  ok"
   ```

4. Test the running service. In the terminal, use the `curl` command to send a JSON query with the input argument 4 to the service through port 9900. For more information on constructing JSON requests, see “JSON Representation of MATLAB Data Types” (MATLAB Production Server).

   ```
   curl -v -H Content-Type:application/json -d '{"nargout":1,"rhs":[4]}' \
   "http://hostname:9900/magicarchive/mymagic"
   ```

   The output is:

   ```
   {"lhs":[{"mwdata":[16,5,9,4,2,11,7,14,3,10,6,15,13,8,12,1],\
   "mwsize":[4,4],"mwtype":"double"}]}
   ```

   Note To use `curl` on Windows, use the following syntax:

   ```
   curl -v -H Content-Type:application/json -d "{\"nargout\":1,\"rhs\":[4]}" \
   "http://hostname:9900/magicarchive/mymagic"
   ```

5 To stop the service, use the following command to display the container id.

```
docker ps
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                     N
df7710d69bf0        micro-magic         "/opt/matlabruntime/…"   6 minutes ago       Up 6 minutes        0.0.0.0:9900->9910/tcp    ep
```

Stop the service using the specified container id.

```
docker stop df7710d69bf0
```

Share Docker Image

You can share your Docker image in various ways.

- Push your image to the Docker central registry DockerHub, or to your private registry. This is the most common workflow.
- Save your image as a tar archive and share it with others. This workflow is suitable for immediate testing.

For details about pushing your image to DockerHub or your private registry, consult the Docker documentation.

Save Docker Image as Tar Archive

To save your Docker image as a tar archive, open a system command window, navigate to the Docker context folder, and type the following.

```
docker save micro-magic -o micro-magic.tar
```

This command creates a file named `micro-magic.tar` in the current folder. Set the appropriate permissions (for example, using `chmod`) prior to sharing the tarball with other users.

Load Docker Image from Tar Archive

Load the image contained in the tarball on the end user machine.

```
docker load --input micro-magic.tar
```

Verify that the image is loaded.

```
docker images
```

Run Docker Image

```
docker run --rm -p 9900:9910 micro-magic
```

See Also

`compiler.package.microserviceDockerImage` | `compiler.build.productionServerArchive`

More About

- “Microservice Command Arguments” on page 1-17
- “Create Deployable Archive for MATLAB Production Server” on page 1-2

- “Client Programming” (MATLAB Production Server)
- “Package MATLAB Standalone Applications into Docker Images”
- “JSON Representation of MATLAB Data Types” (MATLAB Production Server)
- “MATLAB Function Signatures in JSON”

Microservice Command Arguments

Use single or double quotes to enclose special characters. For example:

```
docker run --rm -p 9900:9910 yolov4od-microservice --cors-allowed-origins '*' -l trace &
docker run --rm -p 9900:9910 yolov4od-microservice --cors-allowed-origins "*" -l trace &
```

Option	Description	Note
`-a,--archive` *`FILE`*	Path to the deployable archive (`CTF` file).	Do not use this option when calling `docker run`; the deployed archive included in the container is specified in the entry point.
`--attach-cache` *`CACHE`*	Provide information about the external cache.	Specify *`CACHE`* in the format of *`connection`*`:`*`provider`*`:`*`host`*`:`*`port`*.
`--attach-cache-key` *`KEY`*	Optional key for the external cache.	None.
`-c,--config-file=`*`muserve_config`*	Specify a configuration file located in *`matlabroot`*`/bin/glnxa64`.	Default file name is `muserve_config`. File must be in TOML or INI format.
`--cors-allowed-origins "`*`LIST`*`"`	Enable cross-origin resource sharing (CORS) and specify the domain origins that are allowed to access the server.	Specify *`LIST`* as * or a list of comma-separated domain origins.
`--disable-control-c`	Disable keyboard interruption for the server.	Default behavior is to enable keyboard interruption.
`--display,--no-display`	Enable or disable X11 display for worker processes on UNIX systems.	Default behavior is to disable display.
`--enable-discovery,--disable-discovery`	Enable or disable access to the discovery API.	Default behavior is to enable access to the discovery API.
`--enable-http-pipelining,--disable-http-pipelining`	Enable or disable parallel execution of pipelined requests.	Default behavior is to enable parallel pipeline execution.
`--enable-metrics,--disable-metrics`	Enable or disable access to the metrics API.	Default behavior is to enable access to the metrics API.
`--endpoint-root` *`FILE`*	Path to the folder containing server endpoint files.	By default, endpoint files are not generated.
`-h,--help`	Display the microservice command line arguments and exit.	None.
`--hide-matlab-error-stack`	Hide the MATLAB error stack sent to clients.	Default behavior is to send the error stack.
`--http` *`PORT`*	HTTP interface port in the Docker container.	Default port is 9910.
`--http-linger-threshold` *`SIZE`*	Amount of data the server discards after an HTTP error and before the server closes the TCP connection.	Specify *`SIZE`* as an integer followed by an optional size unit. Allowed size units are B, KB, and MB. If you specify no size unit, the unit is assumed to be B. Default size is unlimited.

Option	Description	Note
`--https PORT`	HTTPS interface port in the Docker container. Use this option to enable HTTPS.	Default behavior is to use HTTP. If you use this option, you must also specify `--x509-private-key` and `--x509-cert-chain`.
`-l,--log-severity OPTION`	Level of detail at which to log information to `stdout`.	Specify *`OPTION`* as `error`, `information` (default), or `trace`.
`--log-format OPTION`	Text format for logs written to `stdout`.	Specify *`OPTION`* as `text-plain` (default), `text-json`, or `text-xml`.
`--merge-worker-streams`	Merge worker `stdout` and `stderr` streams into a single stream.	Default behavior is to keep the streams separate.
`--pid-root PATH`	Path to folder containing PID files.	By default, PID files are not generated.
`--profile "(on|off) OBJECT"`	Enable or disable the logging of server profile information to `stdout`.	Specify *`OBJECT`* as `server`, `server.request`, `server.request.archive`, `server.request.client`, `server.worker`, or `server.worker.pool`.
`--request-size-limit SIZE`	Maximum allowed request size.	Specify *`SIZE`* as an integer followed by an optional size unit. Allowed size units are B, KB, MB, and GB. If you specify no size unit, the unit is assumed to be B. Default size is 64MB.
`--routes-file FILE`	Path to the routes JSON file for the web request handler.	None.
`--ssl-allowed-client CLIENT CN`	Authorize clients to access the deployed archive (CTF file) based on the client certificate common name (CN).	Specify *`CLIENT`* as `client1 CN, client2 CN, ..., clientN CN`.
`--ssl-ciphers OPTION`	List of SSL cipher suites used for encryption.	Specify *`OPTION`* as one of the following: • `ALL` (default) — All cipher suites except the eNULL ciphers. • `HIGH` — Cipher suites with key lengths larger than 128 bits, and some cipher suites with 128-bit keys.
`--ssl-protocols PROTOCOLS`	List of allowed SSL protocols.	Protocols supported: TLSv1, TLSv1.1, TLSv1.2.
`--ssl-tmp-dh-param FILE`	Path to file containing a pregenerated ephemeral DH key.	None.
`--ssl-tmp-ec-param ELLIPTIC-CURVE-NAME`	Name of elliptic curve used for ECDHE ciphers.	ECDHE ciphers are enabled by default.
`--ssl-verify-peer-mode OPTION`	Level of client verification required by the server.	Specify *`OPTION`* as `no-verify-peer` (default) or `verify-peer-require-peer-cert`
`--use-single-comp-thread`	Limit MATLAB to a single computational thread.	Default behavior is to use multithreading capabilities of the host computer.
`--user-data "KEY VALUE"`	Associate MATLAB data value with a key.	*`KEY`* and *`VALUE`* are strings.

Option	Description	Note
`--worker-restart-interval` *`INTERVAL`*	Time interval at which a server stops and restarts its workers. Specify interval in the format [hour]:[minute]:[second].[millisecond].	Default interval is 12:00:00.
`--worker-restart-memory-limit` *`SIZE`*	Size threshold at which the server considers restarting a worker.	Specify *`SIZE`* as an integer followed by an optional size unit. Allowed size units are B, KB, and MB. If you specify no size unit, the unit is assumed to be B.
`--worker-restart-memory-limit-interval` *`INTERVAL`*	Time interval for which a worker can exceed its memory limit before restarting. Specify interval in the format [hour]:[minute]:[second].[millisecond].	None.
`--x509-ca-file-store` *`FILE`*	Path to certificate authority (CA) file to verify peer certificates.	None.
`--x509-cert-chain` *`FILE`*	Path to server certificate chain file in PEM format.	You must specify this property if you specify `--https`.
`--x509-passphrase` *`FILE`*	Path to file that contains the passphrase of the encrypted private key.	None.
`--x509-private-key` *`FILE`*	Path to the private key. The key must be in PEM format.	You must specify this property if you specify `--https`.
`--x509-use-crl`	Use the certificate revocation list (CRL) from the certificate authority store.	None.
`--x509-use-system-store`	Use the operating system truststore.	None.

See Also

Related Examples

- “Create Microservice Docker Image” on page 1-12

Deploy Object Detection Model as Microservice

Supported platform: Linux, Windows, macOS

This example shows how to create a microservice Docker image from a MATLAB object detection model. The microservice image created by MATLAB Compiler SDK provides an HTTP/HTTPS endpoint to access MATLAB code.

You package a MATLAB function into a deployable archive, and then create a Docker image that contains the archive and a minimal MATLAB Runtime package. You can then run the image in Docker and make calls to the service using any of the MATLAB Production Server client APIs.

Required Products

Type `ver` at the MATLAB command prompt to verify whether the following products are installed:

- MATLAB
- Image Processing Toolbox™
- Deep Learning Toolbox™
- Computer Vision Toolbox™
- MATLAB Compiler™
- MATLAB Compiler SDK

Type `matlabshared.supportpkg.getInstalled` at the MATLAB command prompt to verify whether the following add-on is installed:

- Computer Vision Toolbox Model for YOLO v4 Object Detection

If you need to install the add-on, click the **Add-Ons** icon in the MATLAB toolstrip and search for the add-on. You can also download and install it from the MathWorks File Exchange.

Prerequisites

- Verify that you have MATLAB Compiler SDK installed on the development machine.
- Verify that you have Docker installed and configured on the development machine by typing `[~,msg] = system('docker version')` in a MATLAB command window.

  Note If you are using WSL, use the command `[~,msg] = system('wsl docker version')` instead.

  If you do not have Docker installed, follow the instructions on the Docker website to install and set up Docker.

  `docs.docker.com/engine/install/`

  - To build microservice images on Windows, you must install either Docker Desktop or Docker on Windows Subsystem for Linux v2 (WSL2). To install Docker Desktop, see `docs.docker.com/desktop/windows/install/`.

For instructions on how to install Docker on WSL2, see `https://www.mathworks.com/matlabcentral/answers/1758410-how-do-i-install-docker-on-wsl2`.

- If the computer you are using is not connected to the Internet, you must download the MATLAB Runtime installer for Linux from a computer that is connected to the Internet and transfer the installer to the computer that is not connected to the Internet. Then, on the offline machine, run the command `compiler.runtime.createInstallerDockerImage(filepath)`, where `filepath` is the path to the MATLAB Runtime installer archive.

  You can download the installer from the MathWorks website.

  `https://www.mathworks.com/products/compiler/matlab-runtime.html`

Create MATLAB Function to Detect Objects

For this example, write an object detection function named `cvt.m` using the following code.

```
function [bboxes, scores, labels] = cvt(imageUrl)
iminfo = imfinfo(imageUrl);
    % Read image
    % If indexed image, read colormap and convert to rgb
    if strcmp(iminfo.ColorType,'indexed') == 1
        [im, cmap] = webread(imageUrl, 'Timeout', 10);
        im = ind2rgb(im, cmap);
    else
        im = webread(imageUrl, 'Timeout', 10);
    end
% Add  pretrained YOLO v4 dataset tinyYOLOv4COCO.mat to MATLAB path for testing
% Comment or remove the next 2 lines of code prior to deploying as microservice
detectorPath = [matlabshared.supportpkg.getSupportPackageRoot, '/toolbox/vision/supportpackages/y
addpath(detectorPath)
load('tinyYOLOv4COCO.mat', 'detector');

% Detect objects in image using detector
[bboxes,scores,labels] = detect(detector,im);
labels = cellstr(labels);
```

Test the function from the MATLAB command line:

```
%% Specify image URL
imageUrl = "https://www.mathworks.com/help/examples/deeplearning_shared/win64/TrafficSignDetectio
%% Display image
imageFile = "trafficimage.jpg";
imageFileFullPath = websave(imageFile, imageUrl);
[im, cmap] = imread(imageFileFullPath);
imshow(im, cmap)
%% Detect objects in image
[bboxes, scores, labels] = cvt(imageUrl)
```

```
bboxes =
  2×4 single matrix
  445.3871  326.4009  223.3270   98.7086
  504.2861  271.4571   45.7471   41.0955
scores =
  2×1 single column vector
    0.9151
    0.6610
```

```
labels =
  2×1 cell array
    {'truck'    }
    {'stop sign'}
```

Create Deployable Archive

Caution Comment the following lines of code in the `cvt.m` file prior to creating a deployable archive.

```
% detectorPath = [matlabshared.supportpkg.getSupportPackageRoot, '/toolbox/vision/supportpackages
% addpath(detectorPath)
```

Package the `cvt` function into a deployable archive using the `compiler.build.productionServerArchive` function.

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

```
buildResults = compiler.build.productionServerArchive('cvt.m', ...
    'ArchiveName','yolov4od','Verbose',true, ...
    'SupportPackages',{'Computer Vision Toolbox Model for YOLO v4 Object Detection'});
```

```
buildResults =
  Results with properties:

                  BuildType: 'productionServerArchive'
                      Files: {'/home/mluser/work/yolov4odproductionServerArchive/yolov4od.ctf'}
    IncludedSupportPackages: {'Computer Vision Toolbox Model for YOLO v4 Object Detection'}
                    Options: [1×1 compiler.build.ProductionServerArchiveOptions]
```

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

Once the build is complete, the function creates a folder named `yolov4odproductionServerArchive` in your current directory to store the deployable archive.

Package Archive into Microservice Docker Image

- Build the microservice Docker image using the `buildResults` object that you created.

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

  ```
  compiler.package.microserviceDockerImage(buildResults,...
      'ImageName','yolov4od-microservice',...
      'DockerContext',fullfile(pwd,'microserviceDockerContext'));
  ```

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

  - `applicationFilesForMATLABCompiler/yolov4od.ctf` — Deployable archive file.
  - `Dockerfile` — Docker file that specifies Docker run-time options.
  - `GettingStarted.txt` — Text file that contains deployment information.

Test Docker Image

1. In a system command window, verify that your `yolov4od-microservice` image is in your list of Docker images.

   ```
   docker images
   ```

   ```
   REPOSITORY                                          TAG           IMAGE ID            CREATED
   yolov4od-microservice                               latest        4401fa2bc057        33 seconds
   matlabruntime/r2023a/update0/4200000000000000       latest        5259656e4a32        24 minutes
   ```

2. Run the `yolov4od-microservice` microservice image from the system command prompt.

   ```
   docker run --rm -p 9900:9910 yolov4od-microservice -l trace &
   ```

   Port 9910 is the default port exposed by the microservice within the Docker container. You can map it to any available port on your host machine. For this example, it is mapped to port 9900.

   You can specify additional options in the Docker command. For a complete list of options, see “Microservice Command Arguments” on page 1-17.

3. Once the microservice container is running in Docker, you can check the status of the service by going to the following URL in a web browser:

   `http://`*`hostname`*`:9900/api/health`

   If the service is ready to receive requests, you see the following message:

   ```
   "status:  ok"
   ```

4. Test the running service. In the terminal, use the `curl` command to send a JSON query with the input argument 4 to the service through port 9900. For more information on constructing JSON requests, see “JSON Representation of MATLAB Data Types” (MATLAB Production Server).

   ```
   curl -v -H Content-Type:application/json \
   -d '{"nargout":3,"rhs":["https://www.mathworks.com/help/examples/deeplearning_shared/win64/Tr
   "http://hostname:9900/yolov4od/cvt" | jq -c
   ```

   The output is:

   ```
   {"lhs":[{"mwdata":[445.387146,504.286102,326.40094,271.457092,223.327026,45.7471,98.7086487,4
   {"mwdata":[0.91510725,0.661022],"mwsize":[2,1],"mwtype":"single"},
   {"mwdata":[{"mwdata":["truck"],"mwsize":[1,5],"mwtype":"char"},
   {"mwdata":["stop sign"],"mwsize":[1,9],"mwtype":"char"}],"mwsize":[2,1],"mwtype":"cell"}]}
   ```

   You can also test from the MATLAB desktop:

   ```
   %% Import MATLAB HTTP interface packages
   import matlab.net.*
   import matlab.net.http.*
   import matlab.net.http.fields.*

   %% Setup message body
   body = MessageBody;
   body.Payload = ...
       '{"nargout": 3,"rhs": ["https://www.mathworks.com/help/examples/deeplearning_shared/win64

   %% Setup request
   requestUri = URI('http://hostname:9900/yolov4od/cvt');
   options = matlab.net.http.HTTPOptions('ConnectTimeout',20,...
       'ConvertResponse',false);
   ```

```
request = RequestMessage;
request.Header = HeaderField('Content-Type','application/json');
request.Method = 'POST';
request.Body = body;

%% Send request & view raw response
response = request.send(requestUri, options);
disp(response.Body.Data)

%% Decode JSON
lhs = mps.json.decoderesponse(response.Body.Data);

%% Clean up printed output
for i = 1:length(lhs)
    [r,c] = size(lhs{i});
    if ~iscell(lhs{i}) && c==1
        tmp(:,i) = num2cell(lhs{i});
    elseif ~iscell(lhs{i}) && c~=1
        tmp(:,i) = num2cell(lhs{i},2);
    else
        tmp(:,i) = lhs{i};
    end
end
%% Display response as a table
T = cell2table(tmp,'VariableNames',{'Boxes', 'Scores', 'Labels'})
```

The output is:

```
T =

  2×3 table

                  Boxes                    Scores        Labels
    ___________________________________    ______    _____________

    445.39     326.4    223.33    98.709    0.91511    {'truck'    }
    504.29    271.46    45.747    41.096    0.66102    {'stop sign'}
```

5 To stop the service, use the following command to display the container id.

```
docker ps
```

```
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS              PORTS
f372b8b574e8        yolov4od-microservice  "/opt/matlabruntime/…"   6 hours ago         Up 6 hours          0.0.0.0:9900->9910/tc
```

Stop the service using the specified container id.

```
docker stop f372b8b574e8
```

Share Docker Image

You can share your Docker image in various ways.

- Push your image to the Docker central registry DockerHub, or to your private registry. This is the most common workflow.
- Save your image as a tar archive and share it with others. This workflow is suitable for immediate testing.

For details about pushing your image to DockerHub or your private registry, consult the Docker documentation.

Save Docker Image as Tar Archive

To save your Docker image as a tar archive, open a system command window, navigate to the Docker context folder, and type the following.

```
docker save yolov4od-microservice -o yolov4od-microservice.tar
```

This command creates a file named `yolov4od-microservice.tar` in the current folder. Set the appropriate permissions (for example, using `chmod`) prior to sharing the tarball with other users.

Load Docker Image from Tar Archive

Load the image contained in the tarball on the end user machine.

```
docker load --input yolov4od-microservice.tar
```

Verify that the image is loaded.

```
docker images
```

Run Docker Image

```
docker run --rm -p 9900:9910 yolov4od-microservice
```

See Also

`compiler.package.microserviceDockerImage` | `compiler.build.productionServerArchive`

More About

- “Create Deployable Archive for MATLAB Production Server” on page 1-2
- “Client Programming” (MATLAB Production Server)
- “JSON Representation of MATLAB Data Types” (MATLAB Production Server)
- “MATLAB Function Signatures in JSON”
- “Package MATLAB Standalone Applications into Docker Images”

Write Deployable MATLAB Code

MATLAB Coding Guidelines

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

- functions cannot depend on nor change MATLAB state.

  Functions deployed with MATLAB Production Server may not always execute on the same instance of the MATLAB Runtime. Each worker access a different MATLAB Runtime instance.
- explicitly use `varargin` and `varargout` for functions with variable inputs and outputs.
- avoid MATLAB figure or GUI code.

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

See Also

More About

- “State-Dependent Functions” on page 2-3
- “Write Deployable MATLAB Code”

State-Dependent Functions

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

Does My MATLAB Function Carry State?

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

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

Defensive Coding Practices

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

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

The following are examples of "defensive coding" practices:

Reset System-Generated Values in the Deployed Application

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

Validate Global or Persistent Variable Values

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

Ensure Access to Data Caches

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

Use Simple Data Types When Possible

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

Avoid Using MATLAB Callback Functions

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

Techniques for Preserving State

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

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

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

Deploying MATLAB Functions Containing MEX Files

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

Coordinate with your server administrator and application developer as needed.

Supported MATLAB Data Types for Client and Server Marshaling

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

Supported Data Types

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

Partially Supported Data Types

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

Unsupported Data Types

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

- MATLAB function handles
- Sparse matrices
- Tables
- Timetables

See Also

More About

- “JSON Representation of MATLAB Data Types” (MATLAB Production Server)

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" (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” (MATLAB Production Server) and “Deploy Archive to MATLAB Production Server” (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”
- “Create Deployable Archive for MATLAB Production Server” (MATLAB Production Server)
- “Run MATLAB Parallel Server and MATLAB Production Server on Azure” (MATLAB Production Server)

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" (MATLAB Production Server) 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" on page 3-5

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” (MATLAB Production Server) 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 , 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” (MATLAB Production Server).

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”.

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 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” (MATLAB Production Server). 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`.

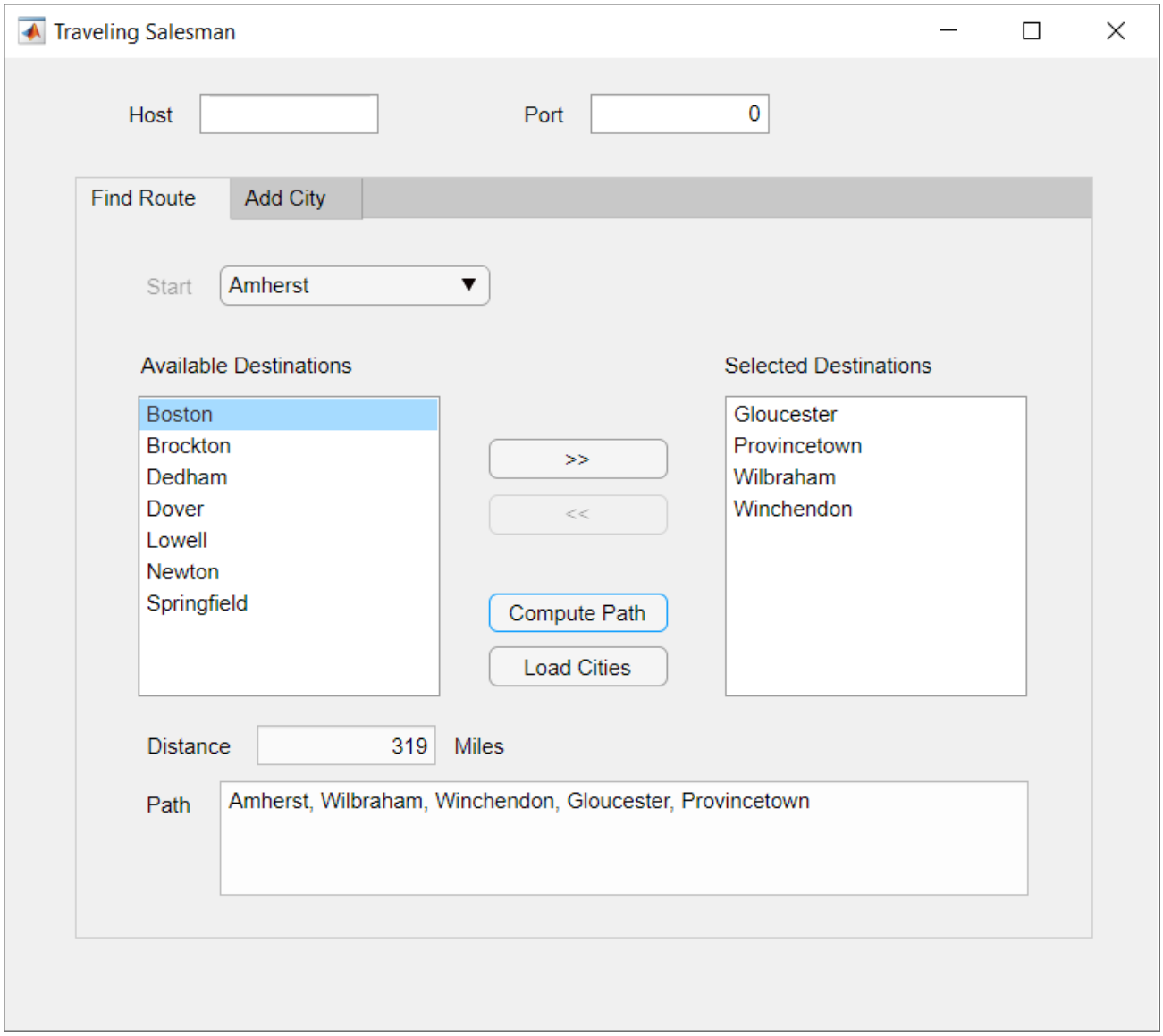
   ```
   >> start(ctrl);
   ```

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.



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 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" (MATLAB Production Server). 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" (MATLAB Production Server).

2. Create a persistence service connection

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

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

   1. Open **Production Server Compiler** app

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

      - MATLAB command prompt: Enter `productionServerCompiler`.

   2. In the **Application Type** menu, select **Deployable Archive**.
   3. In the **Exported Functions** field, add `findRoute.m`, `listDestinations.m` and `addDestination.m`.
   4. Under **Archive information**, rename the archive to `TravelingSalesman`.
   5. Under **Additional files required for your archive to run**, add `Distances.xlsx`.
   6. Click **Package**.
   7. The generated deployable archive `TravelingSalesman.ctf` is located in the `for_redistribution` folder of the project. Copy the `TravelingSalesman.ctf` file to the `auto_deploy` folder of the server, `/tmp/server_1/auto_deploy` in this example, for hosting.

4. Start the server instance

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

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

   Alternatively, use the dashboard to start the server.

5. Start the persistence service

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

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

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

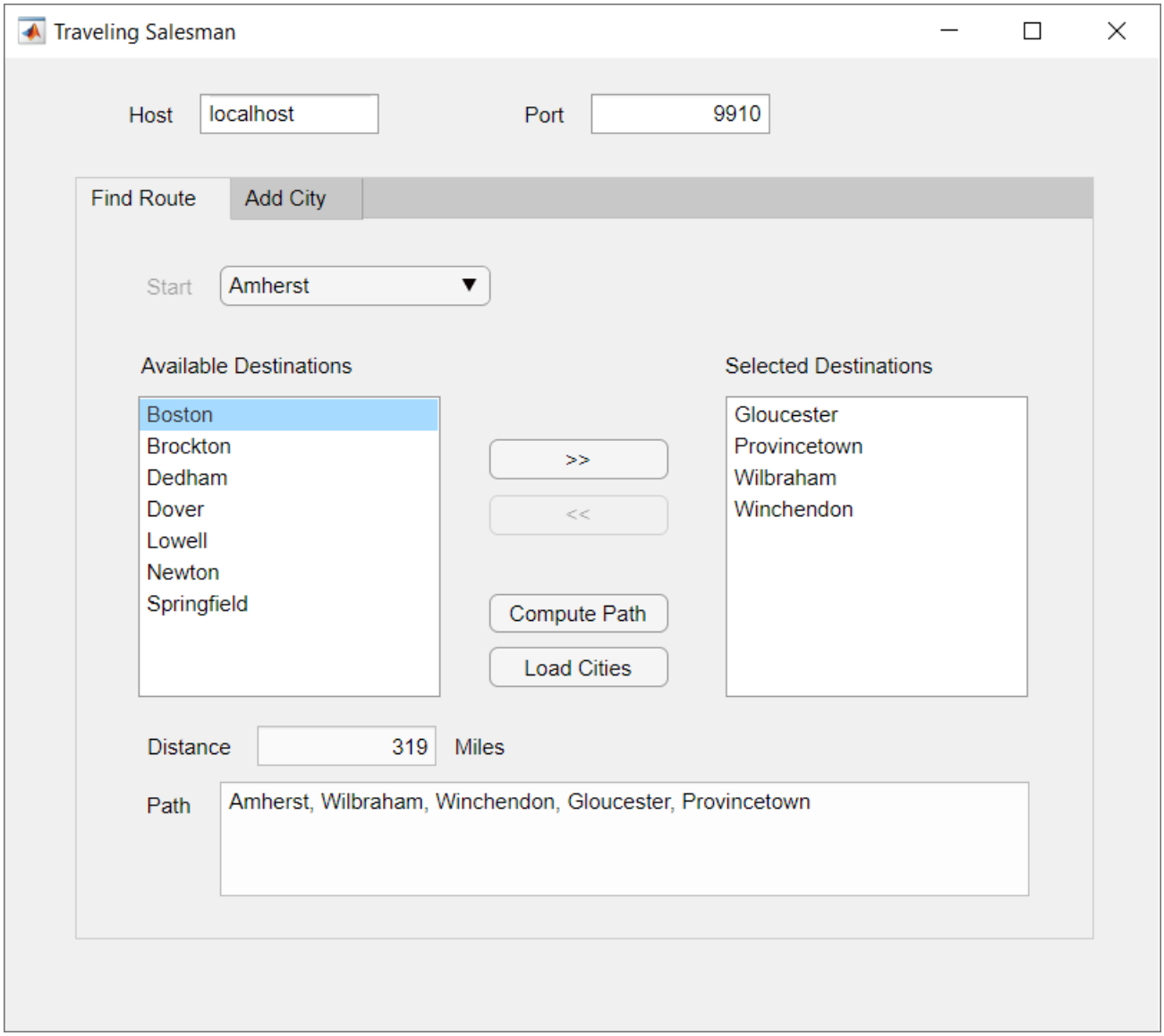
6. Test the app

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

   ```
   >> TravelingSalesman
   ```

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

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



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

See Also

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

More About

- “Data Caching Basics” on page 3-2

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” (MATLAB Production Server).
- 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” (MATLAB Production Server).

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” (MATLAB Production Server).

   ```
   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” (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 (MATLAB Production Server).

   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” on page 4-12
- “Create Deployable Archive for MATLAB Production Server” (MATLAB Production Server)
- “Deploy Archive to MATLAB Production Server” (MATLAB Production Server)

MATLAB Production Server Integration Testing

Write a Test Client

Integration testing with the MATLAB embedded server instance requires a client that calls the compiled MATLAB functions. The client can be coded using any of the MATLAB Production Server client APIs.

At a minimum, the client must:

1. Instantiate the client runtime.
2. Connect to the embedded server instance using the port specified in the Production Server Compiler app.
3. Call the functions being tested with appropriate data.

For information on writing client code, see:

- “Create MATLAB Production Server Java Client Using MWHttpClient Class” (MATLAB Production Server)
- “Create a C# Client” (MATLAB Production Server)
- “Create a Python Client” (MATLAB Production Server)
- “Create a C++ Client” (MATLAB Production Server)

Test Client Data Integration Against MATLAB

In this section...

This example shows how to test your RESTful API or Java client for deployment to MATLAB Production Server using the development version of MATLAB Production Server. MATLAB Compiler SDK includes the development version of MATLAB Production Server for testing and debugging application code and Excel add-ins before deploying them to web applications and enterprise systems.

For testing purposes, you will create and use a MATLAB function called `addmatrix` that accepts two numeric matrices as inputs and returns their sum as an output. You access the local test server by clicking the **Test Client** button in the **Production Server Compiler** app.

Create a MATLAB Function

1. Write a MATLAB function called `addmatrix` that accepts two numeric matrices as inputs and returns their sum as an output. Save this file as `addmatrix.m`.

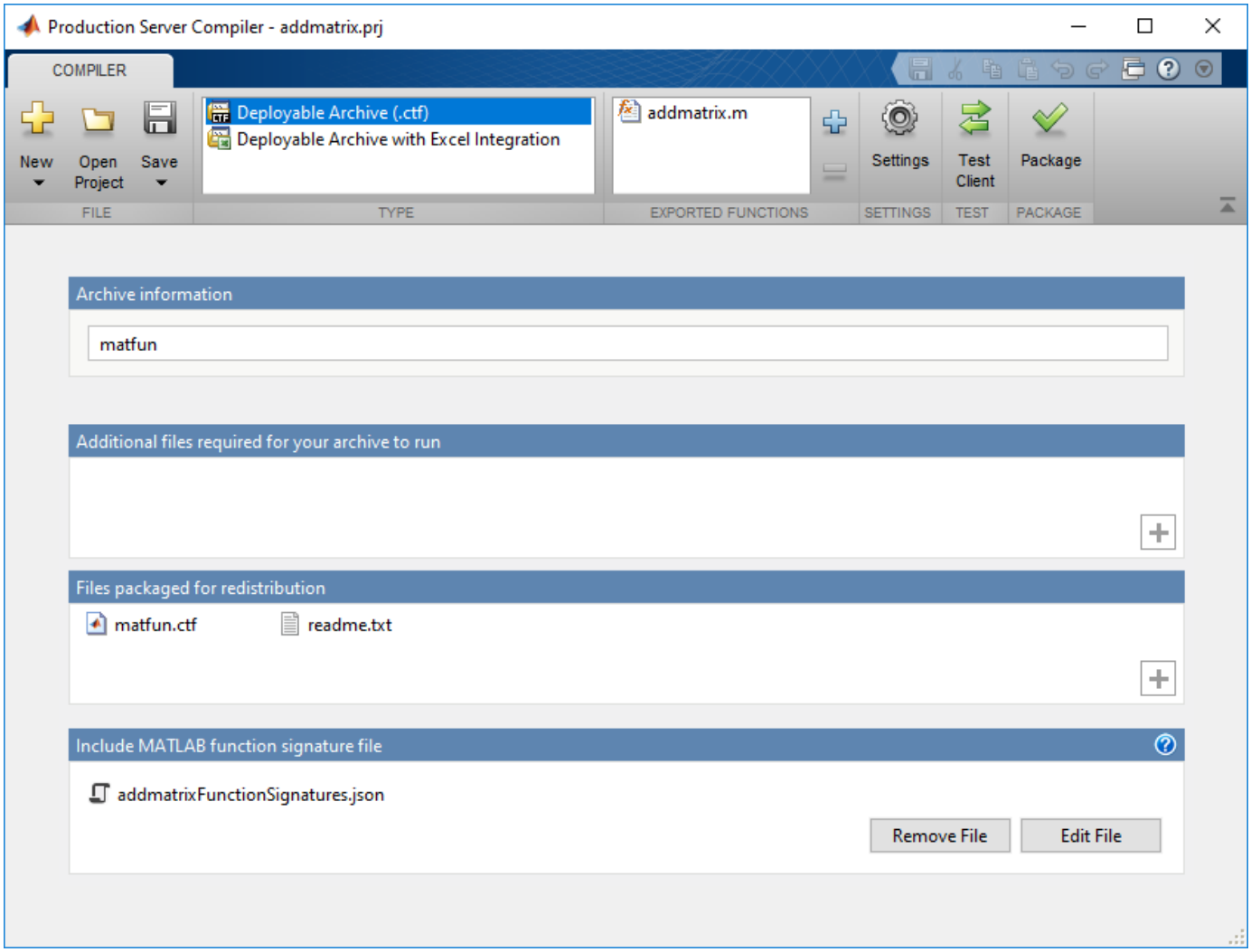
   ```
   function a = addmatrix(a1, a2)
   a = a1 + a2;
   ```
2. Test the function at the MATLAB command prompt.

   ```
   a = [10 20 30; 40 50 60];
   b = [100 200 300; 400 500 600];
   c = addmatrix(a,b)

   c =

      110   220   330
      440   550   660
   ```

Prepare for Testing

1. Open the **Production Server Compiler** app by typing the following at the MATLAB command prompt:

   ```
   productionServerCompiler
   ```



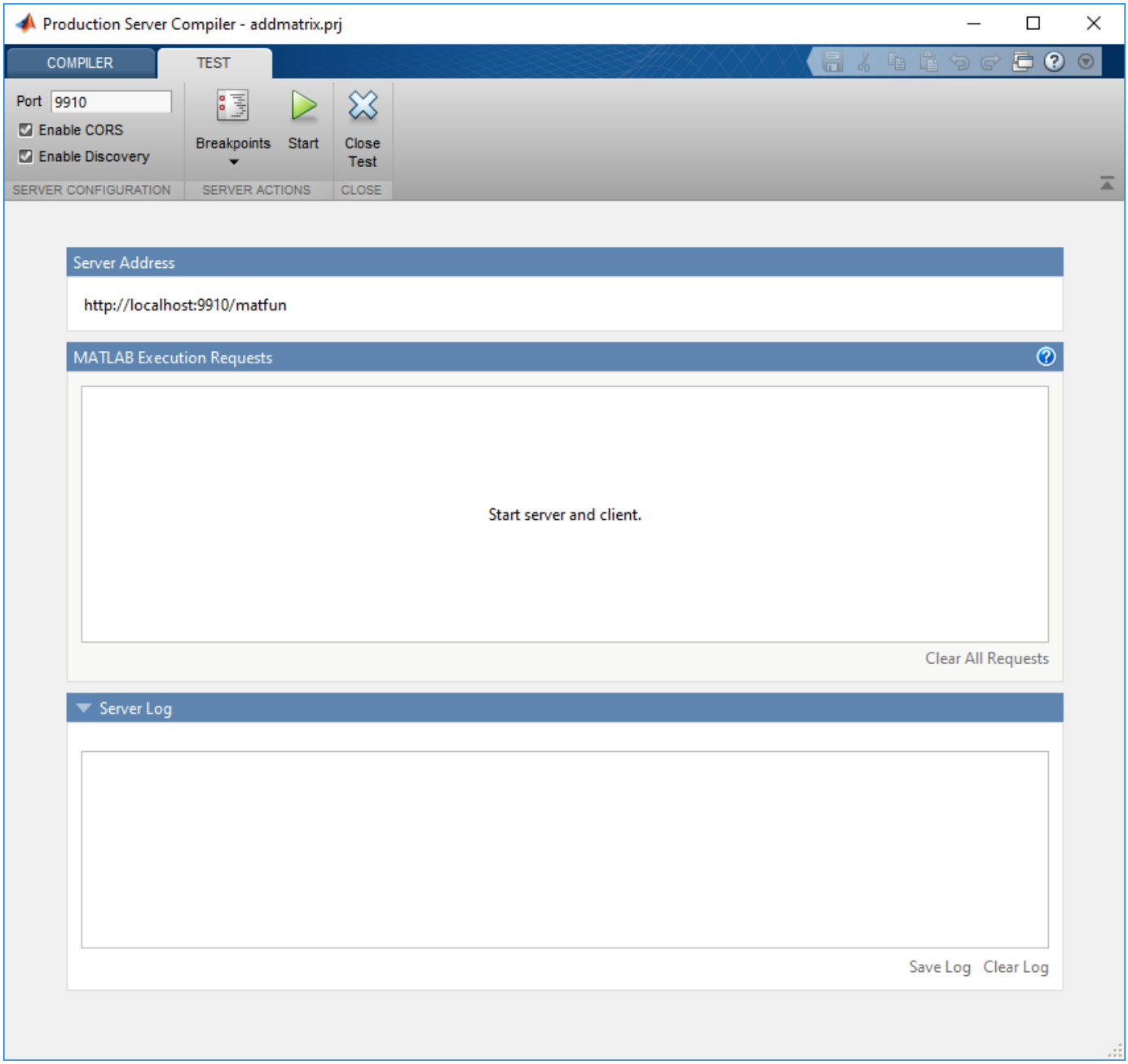
2. In the **Type** section of the toolstrip, select **Deployable Archive (.ctf)** from the list.
3. Specify the MATLAB functions to deploy.
   a. In the **Exported Functions** section of the toolstrip, click the plus button.
   b. Using the file explorer, locate and select the `addmatrix.m` file.
4. In the section titled **Include MATLAB function signature file**, click the **Create File** button. This will create an editable JSON file that contains the function signatures of the functions included in the archive. By editing this file you can specify argument types and/or sizes of inputs and outputs, and also provide help information for each of the inputs. For more information, see "MATLAB Function Signatures in JSON" (MATLAB Production Server).

   If you have an existing JSON file with function signatures, click the **Add Existing File** button to add that file instead of the **Create File** button.

   By including this information in your archive, you can use the discovery service functionality on the server.

   Note Only the MATLAB Production Server RESTful API supports the discovery service. For more information, see "RESTful API for MATLAB Function Execution" (MATLAB Production Server).

5 Click the **Test Client** button. The app will switch to the **TEST** tab.



a Check the value of the **Port** field.

It must be:

- an available port
- the same port number the client is using

For this example, the client will use port 9910.

b Check the box to **Enable CORS**. This option needs to be enabled if you are using a client that uses JavaScript®. By enabling CORS the server will accept requests from different domains.

c Check the box to **Enable Discovery**. This option needs to be enabled to use the discovery service. The discovery service returns information about deployed MATLAB functions as a JSON object.

6 Click **Start**.

Test Using RESTful API

This example uses the MATLAB "Use HTTP with MATLAB" to invoke the RESTful API and make requests to the testing interface. You can use other tools such cURL or JavaScript XHR.

The testing interface does not support asynchronous client requests. The interface processes a POST Asynchronous Request (MATLAB Production Server) like a POST Synchronous Request (MATLAB Production Server). Other asynchronous requests from the RESTful API are not supported.

Test Discovery Service

1 Import the MATLAB HTTP Interface packages, setup the request, and send the request to the testing interface.

```
% Import MATLAB HTTP Interface packages
import matlab.net.*
import matlab.net.http.*
import matlab.net.http.fields.*

% Setup request
requestUri = URI('http://localhost:9910/api/discovery');
options = matlab.net.http.HTTPOptions('ConnectTimeout',20,...
    'ConvertResponse',false);
request = RequestMessage;
request.Header = HeaderField('Content-Type','application/json');
request.Method = 'GET';

% Send request
response = request.send(requestUri, options);
```

2 View the response body.

```
response.Body.Data

ans =

    "{"discoverySchemaVersion":"1.0.0","archives":{"matfun":{"archiveSchemaVersion":"1.1.0",.
```

The response body has been snipped to fit the page. A formatted version of the response body can be found by expanding `ans`.

`ans`

```
{
  "discoverySchemaVersion": "1.0.0",
  "archives": {
    "matfun": {
      "archiveSchemaVersion": "1.1.0",
      "archiveUuid": "",
```

```
      "functions": {
        "addmatrix": {
          "signatures": [
            {
              "help": "",
              "inputs": [
                {
                  "help": "input matrix 1",
                  "mwsize": [],
                  "mwtype": "double",
                  "name": "a1"
                },
                {
                  "help": "input matrix 2",
                  "mwsize": [],
                  "mwtype": "double",
                  "name": "a2"
                }
              ],
              "outputs": [
                {
                  "help": "output matrix",
                  "mwsize": [],
                  "mwtype": "double",
                  "name": "a"
                }
              ]
            }
          ]
        }
      },
      "matlabRuntimeVersion": "9.6.0"
    }
  }
}
```

To test using JavaScript XHR you can use the following code:

JavaScript XHR Code for Testing Discovery Service

```
var data = null;
var xhr = new XMLHttpRequest();
xhr.addEventListener("readystatechange", function () {
  if (this.readyState === 4) {
    console.log(this.responseText);
  }
});
xhr.open("GET", "http://localhost:9910/api/discovery");
xhr.send(data);
```

Testing Data Exchange

1 Start a separate session of the MATLAB desktop.

Note You must use a separate MATLAB session to make POST requests. If you make POST requests from the same MATLAB session that is running the testing interface, MATLAB does not respond.

---

2 Import the MATLAB HTTP Interface packages, setup the request, and send the request to the testing interface.

```
% Import HTTP interface packages
import matlab.net.*
import matlab.net.http.*
import matlab.net.http.fields.*

% Setup message body
body = MessageBody;
a = [10 20 30; 40 50 60];
b = [100 200 300;400 500 600];
payload = mps.json.encoderequest({a,b});
body.Payload = payload;

% Setup request
requestUri = URI('http://localhost:9910/matfun/addmatrix');
options = matlab.net.http.HTTPOptions('ConnectTimeout',20,...
    'ConvertResponse',false);
request = RequestMessage;
request.Header = HeaderField('Content-Type','application/json');
request.Method = 'POST';
request.Body = body;

% Send request
response = request.send(requestUri, options)
```

3 View the response body.

```
response.Body.Data

ans =

    "{"lhs":[[[110,220,330],[440,550,660]]]}"
```

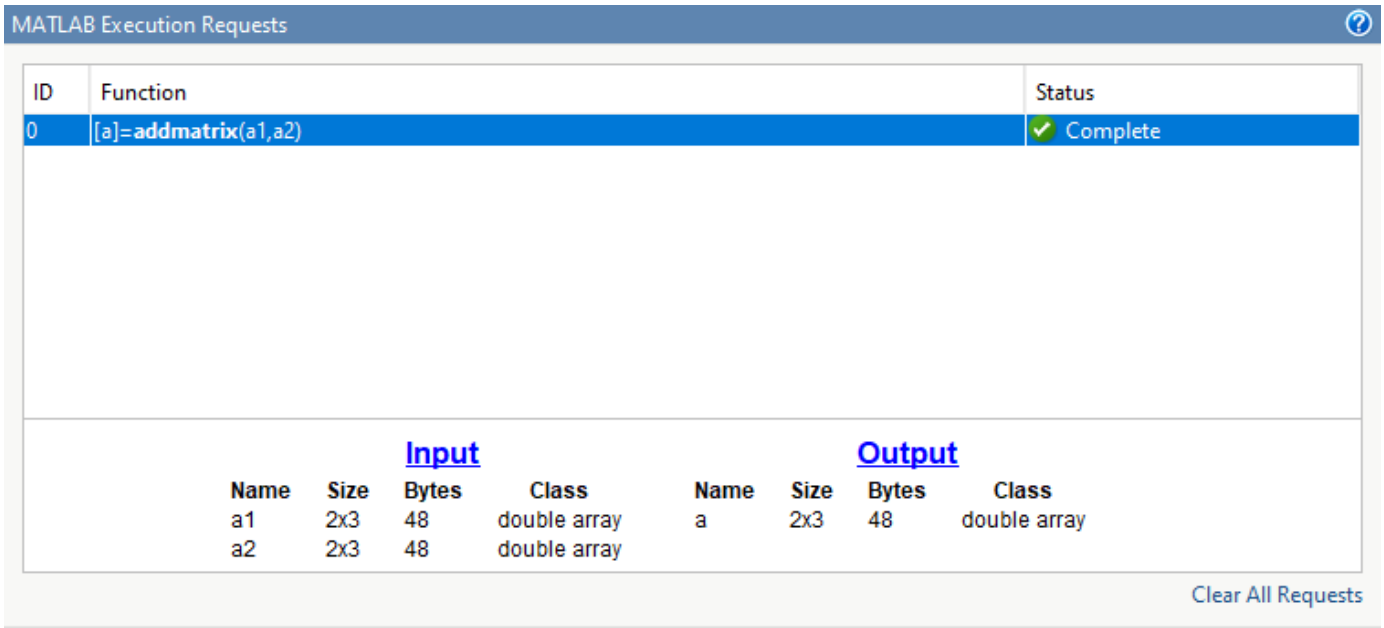
To test using JavaScript XHR you can use the following code:

JavaScript XHR Code for Testing Data Exchange

```
var data = JSON.stringify({
  "rhs": [[[10,20,30],[40,50,60]],[[100,200,300],[400,500,600]]],
  "nargout": 1,
  "outputFormat": {
    "mode": "small",
    "nanType": "string"
  }
});
var xhr = new XMLHttpRequest();
xhr.addEventListener("readystatechange", function () {
  if (this.readyState === 4) {
    console.log(this.responseText);
  }
});
xhr.open("POST", "http://localhost:9910/matfun/addmatrix");
xhr.setRequestHeader("Content-Type", "application/json");
xhr.send(data);
```

Examine Data

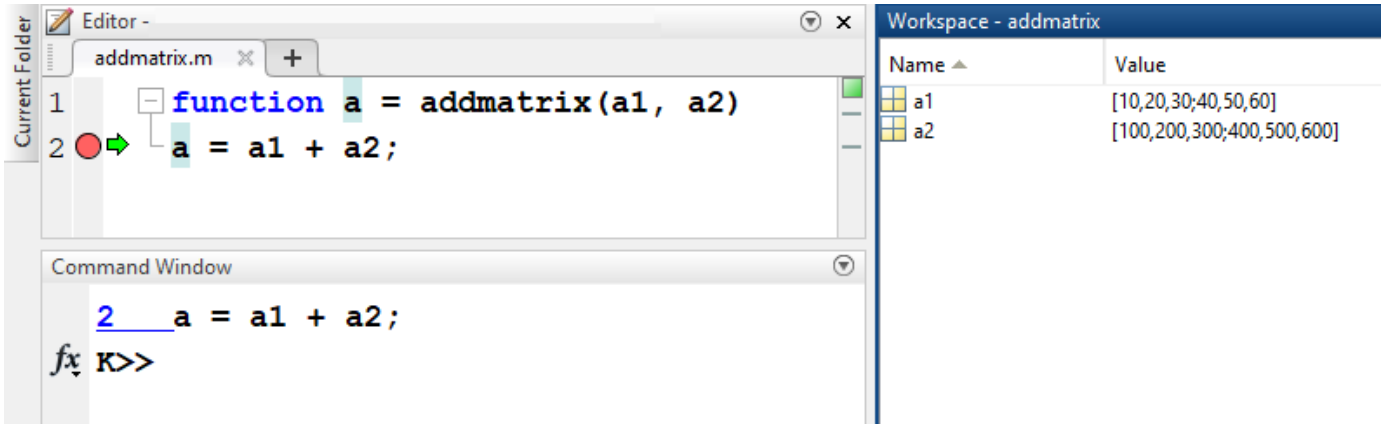
1. Switch to the **Production Server Compiler** app.



2. In the testing interface, under **MATLAB Execution Requests**, click the completed message in the app to see the values exchanged between the client and MATLAB.
3. Click **Input** to view the arrays passed into MATLAB.
4. Click **Output** to view the array returned to the client.

Set Breakpoints

1. In the testing interface of the **Production Server Compiler**, click **Breakpoints > Break on MATLAB function entry**.
2. In the separate MATLAB session, resend a POST request to the local test server.
3. When the MATLAB editor opens, note that a breakpoint is set at the first line in the function and that processing has paused at the breakpoint.



You now can use all of the MATLAB debugging tools to step through your function.

Note You can create a timeout error in the client if you take a long time stepping through the MATLAB function.

4 Note that variables `a1` and `a2` are displayed in the MATLAB workspace.
5 In the MATLAB editor, click **Continue** to complete the debug process.

The **Server Requests** section of the app shows that the request completed successfully.
6 Click **Stop** to shutdown the local test server.
7 Click **Close Test**.

Testing Using Java Client Application

1 Create a Java file `MPSClientExample.java` with following client code:

MPSClientExample.java

```
import java.net.URL;
import java.io.IOException;
import com.mathworks.mps.client.MWClient;
import com.mathworks.mps.client.MWHttpClient;
import com.mathworks.mps.client.MATLABException;

interface MATLABAddMatrix
  {
    double[][] addmatrix(double[][] a1, double[][] a2)
         throws MATLABException, IOException;
  }

public class MPSClientExample {

    public static void main(String[] args){

        double[][] a1={{1,2,3},{3,2,1}};
        double[][] a2={{4,5,6},{6,5,4}};

        MWClient client = new MWHttpClient();

        try{
            MATLABAddMatrix m = client.createProxy(new URL("http://localhost:9910/addmatrix"),
                                                   MATLABAddMatrix.class);
            double[][] result = m.addmatrix(a1,a2);

            // Print the magic square

            printResult(result);

        }catch(MATLABException ex){

            // This exception represents errors in MATLAB
               System.out.println(ex);
        }catch(IOException ex){

            // This exception represents network issues.
               System.out.println(ex);
        }finally{

            client.close();
        }
    }

    private static void printResult(double[][] result){
        for(double[] row : result){
            for(double element : row){
                System.out.print(element + " ");
            }
            System.out.println();
        }
    }
}
```

2 At the system command prompt, compile the Java client code using the `javac` command.

```
javac -classpath "matlabroot\toolbox\compiler_sdk\mps_clients\java\mps_client.jar" MPSClientExample.java
```

3 At the system command prompt, run the Java client.

```
java -classpath .;"matlabroot\toolbox\compiler_sdk\mps_clients\java\mps_client.jar" MPSClientExample
```

Note You cannot run the Java client from the MATLAB command prompt.

The application returns the following at the console:

```
   110.0   220.0   330.0
   440.0   550.0   660.0
```

You can debug the data exchanged between the client and MATLAB using the same steps listed under “Test Using RESTful API” on page 4-6.

See Also

Related Examples

- “Write a Test Client” on page 4-2
- “Package Deployable Archives with Production Server Compiler App”
- “Test Web Request Handlers” on page 4-12
- “MATLAB Not Responding to Web Requests Made to Test Server” on page 4-17

Test Web Request Handlers

You can use the testing interface in the **Production Server Compiler** app to test web request handlers for deployment to MATLAB Production Server. A web request handler consists of MATLAB functions and a JSON file that specifies URL routes. To set up the testing interface for web request handlers, you configure access to the routes JSON file.

For configuring access to the routes file, either set an environment variable that specifies the path to the routes file or place the routes file in the MATLAB preferences directory. When you start the testing interface, it searches for the environment variable for the routes file first. If the environment variable is not set, then the testing interface searches the MATLAB preferences directory for the routes file. After you configure access to the routes file, you can test the MATLAB functions for web request handlers. For more information about web request handlers, see “Handle Custom Routes and Payloads in HTTP Requests” (MATLAB Production Server).

Set Environment Variable for Routes File

Set the environment variable `PRODSERVER_ROUTES_FILE` to a value that contains the path to the routes file. You can set the environment variable at the MATLAB prompt using `setenv` or at the system command prompt using syntax specific to your operating system.

```
setenv('PRODSERVER_ROUTES_FILE', 'path/to/routes/file/routes.json');
```

- If you specify a relative path to the routes file, from the MATLAB prompt, navigate to the folder that contains the routes file before you start the test server in the **Production Server Compiler** app.
- If you update the contents or location of a routes file that is already in use, for your changes to take effect, restart the test server in the **Production Server Compiler** app.
- To turn off testing for web request handlers, set `PRODSERVER_ROUTES_FILE` to an empty value.

Use MATLAB Preferences Folder for Routes File

An alternate option for configuring access to the routes file is to copy the file to the MATLAB preferences folder. This configuration persists across MATLAB restarts. You must name the routes file `prodserver_routes.json`. To locate your preferences folder, type `prefdir` at the MATLAB prompt.

- If you update the contents or location of a routes file that is already in use, for your changes to take effect, restart the test server in the **Production Server Compiler** app.
- To turn off testing for web request handlers, rename or move `prodserver_routes.json` from the preferences folder.

End-to-End Setup to Test Web Request Handlers

Create Routes File

Using a text editor, create a routes JSON file to map client requests to the MATLAB web request handler functions. Save the file as `routes.json`.

The following routes file maps any client request that contains `MyDemo` in the request URL to a `hellowh` MATLAB function in the `whdemo` deployable archive.

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

Configure Access to Routes File

From the MATLAB prompt, set the environment variable `PRODSERVER_ROUTES_FILE` to specify the path to the routes file.

```
setenv('PRODSERVER_ROUTES_FILE', 'J:\routes.json');
```

Write MATLAB Function for Web Request Handler

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

The following code shows a MATLAB function, `hellowh.m`, that uses the input argument structure `request`, whose fields provide information about the request headers and body. The function also constructs and returns the 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
```

Prepare for Testing

1 Open the **Production Server Compiler** app by typing the following at the MATLAB command prompt:

```
productionServerCompiler
```

2. In the **Type** section of the toolstrip, select **Deployable Archive (.ctf)**.
3. Specify the MATLAB functions to deploy.
   a. In the **Exported Functions** section of the toolstrip, click the plus button.
   b. Using the file explorer, locate and select the `hellowh.m` file.
4. Click **Test Client**. The app switches to the **TEST** tab.
5. Click **Start** to start your test. The **Server Log** section displays errors, if any.

Call Web Handler MATLAB Function

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.

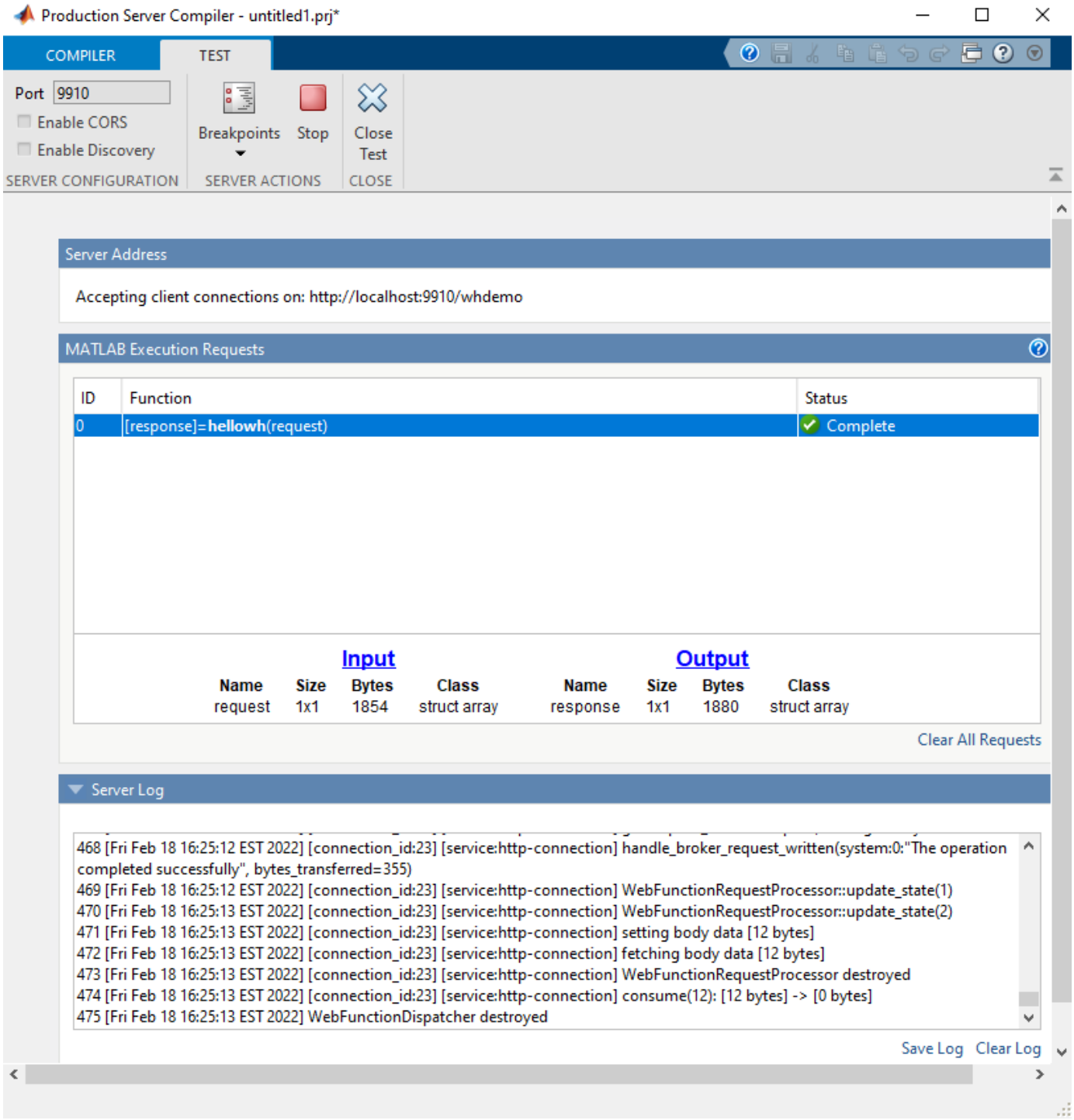
```
curl -v http://localhost:9910/MyDemo/this/could/be/any/path?param=YES
```

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
```

Examine Data

1. Switch back to the **Production Server Compiler** app.
2. In the testing interface, under **MATLAB Execution Requests**, click the completed message in the app to see the values exchanged between the client and MATLAB.



3 Click **Input** to view data passed into MATLAB.

4 Click **Output** to view data returned to the client.

After you are satisfied with your testing, you can package the MATLAB function and deploy it to the server. For more information, see “Create Deployable Archive for MATLAB Production Server” on page 1-2.

See Also

Related Examples

- “Handle Custom Routes and Payloads in HTTP Requests” (MATLAB Production Server)
- “Test Client Data Integration Against MATLAB” on page 4-3

MATLAB Not Responding to Web Requests Made to Test Server

Issue

When testing the integration of client code with MATLAB functions using the **Production Server Compiler** app, if your client code makes web requests from the same MATLAB session as the local test server, MATLAB stops responding. This issue can occur, for example, when making RESTful API calls using functions such as `webread` and `webwrite` or when using the MATLAB HTTP Interface.

Possible Solutions

Open a separate session of MATLAB and make your client web requests to the local test server from the new session.

See Also

Related Examples

- “Test Client Data Integration Against MATLAB” on page 4-3
- “What Is the HTTP Interface?”

5

MATLAB Production Server Excel Add-In

Data Marshaling Rules

In this section...

Default Marshaling Rules

These types of data do not have natural mappings between MATLAB and Excel:

- Dates: Excel has a special data type for dates, and MATLAB does not.
- Blank cells: MATLAB has no equivalent construct for a blank cell in an Excel spread sheet.

If you do not change the marshaling rules when compiling the add-in, the rules for marshaling Excel data into MATLAB are:

- Excel dates are marshaled into MATLAB doubles.
- Empty cells are marshaled into zeros.

If you do not change the marshaling rules when compiling the add-in, the rules for marshaling MATLAB data into Excel are:

- MATLAB NaNs are marshaled into Visual Basic® `#QNAN`s.
- MATLAB does not return any Excel dates.

Change Rules for Marshaling Data into MATLAB

You can change how dates and empty cells are marshaled into MATLAB when compiling the add-in:

- Excel dates can be marshaled as MATLAB character arrays.
- Empty cells can be marshaled as MATLAB NaNs.

To change the marshaling rules:

1. In the class mapper portion of the **MATLAB Compiler** project window, select the signature of the function you want to modify.
2. Select **Data Conversion Properties** from the context menu.
3. Select the input argument rules to change.
4. Click outside of the dialog box to close it.

Change Rules for Marshaling Data into Excel

You can change how dates and NaNs are marshaled into Excel when compiling the add-in:

- MATLAB NaNs can be converted into zeros.
- MATLAB numeric values can be converted into Excel dates.

Note To see a date in the expected format, ensure that the Excel cell is formatted to display its contents in a date format.

To change the marshaling rules:

1. In the class mapper portion of the **MATLAB Compiler** project window, select the signature of the function you want to modify.
2. Select **Data Conversion Properties** from the context menu.
3. Select the output argument rules to change.
4. Click outside of the dialog box to close it.

6

MATLAB Production Server Excel Add-In

XLA File Not Generated

The compiler may not generate the *projName*.xla file for various reasons, including that Excel is not configured to trust access to the VBA project object model. When this happens, you can install the add-in by importing the *projName*.bas file into the workbook's Visual Basic project.

Server Configuration Add-in Not Enabled

If your trust settings in Excel are configured to either disable all add-ins or to require add-ins to be published by a trusted publisher, it is possible that the **Configure MATLAB Production Server** add-in is not available after installation. In most cases, the add-in is installed but disabled.

To check if the add-in is installed in Excel:

1. Select **File>Options**.
2. Select **Add-Ins**.
3. Look for `ServerConfig.Connect` in the list of disabled add-ins.

You can enable the add-in by adjusting the trust settings in Excel.

Error Using a Variable Number of Outputs

If your add-in throws the error:

```
Error in myfunc: Object reference not set to an instance of an object
```

The likely cause is that the MATLAB function used by the add-in returns a variable number of outputs.

Add-ins using code run on a MATLAB Production Server instance do not support MATLAB functions that return a variable number of outputs. You can either rewrite your MATLAB function to return a fixed number of outputs, or you can create an add-in that runs locally to your Excel installation.

Functions

compiler.build.excelClientForProductionServer

Create Microsoft Excel add-in for MATLAB Production Server

Syntax

```
compiler.build.excelClientForProductionServer(Results)
compiler.build.excelClientForProductionServer(FunctionFiles,ServerArchive)
compiler.build.excelClientForProductionServer(FunctionFiles,ServerArchive,
Name,Value)
compiler.build.excelClientForProductionServer(opts)
results = compiler.build.excelClientForProductionServer( ___ )
```

Description

Caution This function is only supported on Windows operating systems.

`compiler.build.excelClientForProductionServer(Results)` creates an Excel add-in for MATLAB Production Server using the `compiler.build.Results` object `Results` created from the `compiler.build.productionServerArchive` function. Before creating Excel add-ins, install a supported compiler.

`compiler.build.excelClientForProductionServer(FunctionFiles,ServerArchive)` creates an Excel add-in using MATLAB functions specified by `FunctionFiles` and the MATLAB Production Server archive specified by `ServerArchive`.

`compiler.build.excelClientForProductionServer(FunctionFiles,ServerArchive, Name,Value)` creates an Excel add-in with options specified using one or more name-value arguments. Options include the add-in name, output directory, and how to handle the Excel date data type.

`compiler.build.excelClientForProductionServer(opts)` creates an Excel add-in with options specified using a `compiler.build.ExcelClientForProductionServerOptions` object `opts`. You cannot specify any other options using name-value arguments.

`results = compiler.build.excelClientForProductionServer( ___ )` 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, paths to the compiled files, and build options.

Examples

Create Excel Add-In Using Results

Create an Excel add-in for MATLAB Production Server on a Windows system using the results from the `compiler.build.productionServerArchive` function.

Ensure that you have the following installed:

- The Windows 10 SDK kit. For details, see Windows 10 SDK.
- MinGW-w64. To install it from the MathWorks File Exchange, see MATLAB Support for MinGW-w64 C/C++ Compiler.

  Use `mbuild -setup -client mbuild_com` to ensure that MATLAB is able to create Excel add-ins.

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

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

Build a MATLAB Production Server archive using the `compiler.build.productionServerArchive` command. Save the output as a `compiler.build.Results` object `serverBuildResults`.

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

Build an Excel add-in for MATLAB Production Server archive using the `compiler.build.excelClientForProductionServer` command.

```
excelBuildResults = compiler.build.excelClientForProductionServer(serverBuildResults);
```

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

- `includedSupportPackages.txt`
- `magicsquare.bas` (Only if you enable the `'GenerateVisualBasicFile'` option)
- `magicsquare.dll`
- `magicsquare.reg`
- `magicsquare.xla` (Only if you enable the `'GenerateVisualBasicFile'` option)
- `magicsquareClass.cs`
- `readme.txt`
- `requiredMCRProducts.txt`

Create Excel Add-In Using Files

Create an Excel add-in for MATLAB Production Server on a Windows system using MATLAB function files and a MATLAB Production Server archive.

Create a MATLAB Production Server archive using a MATLAB function file. For this example, use the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler` as an input to the `compiler.build.productionServerArchive` function.

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

The function generates the file `magicsquare.ctf` in the `magicsquareproductionServerArchive` folder.

Build an Excel add-in for MATLAB Production Server archive using the `compiler.build.excelClientForProductionServer` command. Specify the function file and the CTF file as inputs.

```
excelBuildResults = compiler.build.excelClientForProductionServer(mpsFile,'magicsquareproductionServerArchive\magicsquare.ctf');
```

Customize Excel Add-In

Create an Excel add-in and customize it using name-value arguments.

Create a MATLAB Production Server archive using a MATLAB function file. For this example, use the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler` as an input to the `compiler.build.productionServerArchive` function.

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

Build an Excel add-in for MATLAB Production Server using the `compiler.build.excelClientForProductionServer` command. Use name-value arguments to specify the add-in name, generate a Microsoft Visual Basic file, and enable verbose output.

```
compiler.build.excelClientForProductionServer(mpsFile,...
    'magicsquareproductionServerArchive\magicsquare.ctf',...
    'AddInName','MyMagicSquare',...
    'GenerateVisualBasicFile','on',...
    'Verbose','on');
```

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

- `includedSupportPackages.txt`
- `MyMagicSquare.bas`
- `MyMagicSquare.dll`
- `MyMagicSquare.reg`
- `MyMagicSquare.xla`
- `MyMagicSquareClass.cs`
- `readme.txt`
- `requiredMCRProducts.txt`

Create Multiple Add-Ins Using Options Object

Create multiple Excel add-ins for MATLAB Production Server on a Windows system using a `compiler.build.ExcelClientForProductionServerOptions` object.

Create a MATLAB Production Server archive using a MATLAB function file. For this example, use the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler` as an input to the `compiler.build.productionServerArchive` function.

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

Create an `ExcelClientForProductionServerOptions` object using the file `houdini.m` located in *`matlabroot`*`\extern\examples\compiler`. Use name-value arguments to specify a common output directory, generate a Visual Basic file, and enable verbose output.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','houdini.m');
opts = compiler.build.ExcelClientForProductionServerOptions(appFile,...
    'magicsquareproductionServerArchive\magicsquare.ctf',...
    'OutputDir','D:\Documents\MATLAB\work\MPSExcelAddInBatch',...
```

```
    'GenerateVisualBasicFile','on',...
    'Verbose','on')

opts =

  ExcelClientForProductionServerOptions with properties:

                        AddInName: 'houdini'
                     AddInVersion: '1.0.0.0'
                        ClassName: 'houdiniClass'
                       DebugBuild: off
                    FunctionFiles: {'C:\Program Files\MATLAB\R2023a\extern\examples\compiler\hou
            GenerateVisualBasicFile: on
                    ServerArchive: 'magicsquareproductionServerArchive\magicsquare.ctf'
          ReplaceExcelBlankWithNaN: off
          ConvertExcelDateToString: off
           ReplaceNaNToZeroInExcel: off
    ConvertNumericOutToDateInExcel: off
                           Verbose: on
                         OutputDir: 'D:\Documents\MATLAB\work\MPSExcelAddInBatch'
```

Build the add-in using the `ExcelAddInOptions` object.

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

To create a new add-in using the function file `houdini.m` with the same options, use dot notation to modify the `FunctionFiles` argument of the existing `ExcelAddInOptions` object before running the build function again.

```
appFile2 = fullfile(matlabroot,'extern','examples','compiler','houdini.m');
opts.FunctionFiles = appFile2;
compiler.build.excelClientForProductionServer(opts);
```

By modifying the `FunctionFiles` argument and recompiling, you can create multiple add-ins using the same options object.

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.

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` — Excel add-in build options

`compiler.build.ExcelClientForProductionServerOptions` object

Excel add-in build options, specified as a `compiler.build.ExcelClientForProductionServerOptions` object.

`Results` — Build results object

`Results` object

Build results, specified as a `compiler.build.Results` object. Create the `Results` object by saving the output from the `compiler.build.productionServerArchive` function.

`ServerArchive` — Excel add-in build options

character vector | string scalar

MATLAB Production Server archive deployed on the Production Server, specified as a character vector or a string scalar.

Example: `'mpsArchive.ctf'`

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: `'Verbose','on'`

`AddInName` — Name of Excel add-in
character vector | string scalar

Name of the Excel add-in, specified as a character vector or string scalar. The default name of the generated add-in is the first entry of the `FunctionFiles` argument. The name must begin with a letter and contain only alphabetic characters and underscores.

Example: `'AddInName','myAddIn'`

Data Types: `char` | `string`

`AddInVersion` — Add-in version
`'1.0.0.0'` (default) | character vector | string scalar

Add-in version, specified as a character vector or a string scalar.

Example: `'AddInVersion','4.0'`

Data Types: `char` | `string`

`ClassName` — Name of class
character vector | string scalar

Name of the generated class, specified as a character vector or a string scalar. You cannot specify this option if you use a `ClassMap` input. Class names must meet the Excel class name requirements.

The default value is the `AddInName` argument appended with `Class`.

Example: `'ClassName','MagicSquareClass'`

Data Types: `char` | `string`

`ConvertExcelDateToString` — Flag to convert date to string
`'off'` (default) | on/off logical value

Flag to convert Excel date to string, 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 compiler converts the Excel date datatype to MATLAB string.
- If you set this property to `'off'`, then dates are not converted.

Example: `'ConvertExcelDateToString','On'`

Data Types: `logical`

`ConvertNumericOutToDateInExcel` — Flag to convert numeric data to Excel date
`'off'` (default) | on/off logical value

Flag to convert numeric data to Excel date, 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 compiler converts numeric data to the Excel date datatype.

- If you set this property to `'off'`, then numeric data is not converted.

Example: `'ConvertNumericOutToDateInExcel','On'`

Data Types: `logical`

`DebugBuild` — Flag to enable debug symbols
`'on'` (default) | on/off logical value

Flag to enable debug symbols, 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 add-in is compiled with debug symbols.
- If you set this property to `'off'`, then the add-in is not compiled with debug symbols.

Example: `'DebugSymbols','On'`

Data Types: `logical`

`GenerateVisualBasicFile` — Flag to generate Visual Basic file
`'off'` (default) | on/off logical value

Flag to generate a Visual Basic file (`.bas`) and an Excel add-in file (`.xla`), 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 function generates an Excel add-in XLA file and a Visual Basic BAS file containing the Microsoft Excel Formula Function interface to the add-in.
- If you set this property to `'off'`, then the function does not generate a Visual Basic file or an Excel add-in file.

Example: `'GenerateVisualBasicFile','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 add-in name appended with `excelAddIn`.

Example: `'OutputDir','D:\Documents\MATLAB\work\mymagicexcelAddIn'`

Data Types: `char` | `string`

`ReplaceExcelBlankWithNaN` — Flag to convert blank Excel cells to NaN
`'off'` (default) | on/off logical value

Flag to convert blank Excel cells to NaN, 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 compiler converts blank Excel cells to NaN in the compiled artifact.
- If you set this property to `'off'`, then blank Excel cells are not converted.

Example: `'ReplaceExcelBlankWithNaN','On'`

Data Types: `logical`

`ReplaceNaNToZeroInExcel` — Flag to convert NaN entries to zero
`'off'` (default) | on/off logical value

Flag to convert NaN entries to zero, 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 compiler converts NaN entries from the compiled artifact to zero in Excel.
- If you set this property to `'off'`, then NaN entries are not converted.

Example: `'ReplaceNaNToZeroInExcel','On'`

Data Types: `logical`

`Verbose` — Flag to control build verbosity
`'off'` (default) | on/off logical value

Flag to control 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','On'`

Data Types: `logical`

Output Arguments

`results` — Build results
`compiler.build.Results` object

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

- Build type, which is `'excelClientForProductionServer'`
- Paths to the following files:
  - *`AddInName`*`.dll`
  - *`AddInName`*`.bas` (if you enable the `'GenerateVisualBasicFile'` option)

  - *AddInName*.xla (if you enable the 'GenerateVisualBasicFile' option)
- A list of included support packages
- Build options, specified as an ExcelClientForProductionServerOptions object

Limitations

- This function is only supported on Windows operating systems.

Version History

Introduced in R2021b

See Also

compiler.build.ExcelClientForProductionServerOptions | compiler.build.Results | **Library Compiler** | mcc

compiler.build.ExcelClientForProductionServerOptions

Options for building Excel add-ins

Syntax

```
opts = compiler.build.ExcelClientForProductionServerOptions(Results)
opts = compiler.build.ExcelClientForProductionServerOptions(FunctionFiles,
ServerArchive)
opts = compiler.build.ExcelClientForProductionServerOptions(FunctionFiles,
ServerArchive,Name,Value)
```

Description

`opts = compiler.build.ExcelClientForProductionServerOptions(Results)` creates an `ExcelClientForProductionServerOptions` object using the `compiler.build.Results` object `Results` created from the `compiler.build.productionServerArchive` function. Use the `ExcelClientForProductionServerOptions` object as an input to the `compiler.build.excelClientForProductionServer` function.

`opts = compiler.build.ExcelClientForProductionServerOptions(FunctionFiles, ServerArchive)` creates an `ExcelClientForProductionServerOptions` object using MATLAB functions specified by `FunctionFiles` and the MATLAB Production Server archive specified by `ServerArchive`.

`opts = compiler.build.ExcelClientForProductionServerOptions(FunctionFiles, ServerArchive,Name,Value)` creates an `ExcelClientForProductionServerOptions` object with options specified using one or more name-value arguments. Options include the add-in name, output directory, and how to handle the Excel date data type.

Examples

Create Excel Add-In Options Object Using Results

Create an `ExcelClientForProductionServerOptions` object using the results from the `compiler.build.productionServerArchive` function.

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

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

Build a MATLAB Production Server archive using the `compiler.build.productionServerArchive` function. Save the output as a `compiler.build.Results` object `serverBuildResults`.

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

Create an `ExcelClientForProductionServerOptions` object using `serverBuildResults` and the `compiler.build.excelClientForProductionServer` function.

```
opts = compiler.build.ExcelClientForProductionServerOptions(serverBuildResults)

opts =

  ExcelClientForProductionServerOptions with properties:

                        AddInName: 'magicsquare'
                     AddInVersion: '1.0.0.0'
                        ClassName: 'magicsquareClass'
                       DebugBuild: off
                    FunctionFiles: {'C:\Program Files\MATLAB\R2023a\extern\
                                   examples\compiler\magicsquare.m'}
          GenerateVisualBasicFile: on
                    ServerArchive: '.\magicsquareproductionServerArchive\magicsquare.ctf'
         ReplaceExcelBlankWithNaN: off
         ConvertExcelDateToString: off
          ReplaceNaNToZeroInExcel: off
    ConvertNumericOutToDateInExcel: off
                          Verbose: off
                        OutputDir: '.\magicsquareexcelClientForProductionServer'
```

Use the `ExcelClientForProductionServerOptions` object as an input to the `compiler.build.excelClientForProductionServer` function to build an Excel add-in for MATLAB Production Server.

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

Create Excel Add-In Options Object Using Files

Create an `ExcelClientForProductionServerOptions` object using a MATLAB function file and a MATLAB Production Server archive.

Build a MATLAB Production Server archive using the `compiler.build.productionServerArchive` function. For this example, use the file `houdini.m` located in *`matlabroot`*`\extern\examples\compiler`.

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

Create an `ExcelClientForProductionServerOptions` object using the MATLAB Production Server archive file `houdini.ctf`.

```
opts = compiler.build.ExcelClientForProductionServerOptions(appFile,...
    'houdiniproductionServerArchive\houdini.ctf')

opts =

  ExcelClientForProductionServerOptions with properties:

                        AddInName: 'houdini'
                     AddInVersion: '1.0.0.0'
                        ClassName: 'houdiniClass'
                       DebugBuild: off
                    FunctionFiles: {'C:\Program Files\MATLAB\R2023a\extern\examples\compiler\hou
          GenerateVisualBasicFile: off
                    ServerArchive: 'houdiniproductionServerArchive\houdini.ctf'
         ReplaceExcelBlankWithNaN: off
```

```
          ConvertExcelDateToString: off
           ReplaceNaNToZeroInExcel: off
    ConvertNumericOutToDateInExcel: off
                           Verbose: off
                         OutputDir: '.\houdiniexcelClientForProductionServer'
```

Use the `ExcelClientForProductionServerOptions` object as an input to the `compiler.build.excelClientForProductionServer` function to build an Excel add-in for MATLAB Production Server.

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

Customize Excel Add-In Options Object

Create an `ExcelClientForProductionServerOptions` object and customize it using name-value arguments.

Build a MATLAB Production Server archive using the `compiler.build.productionServerArchive` function. For this example, use the file `houdini.m` located in *`matlabroot`*`\extern\examples\compiler`.

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

Create an `ExcelClientForProductionServerOptions` object using the MATLAB Production Server archive file `houdini.ctf`. Use name-value arguments to specify the output directory and generate a Visual Basic file.

```
opts = compiler.build.ExcelClientForProductionServerOptions(appFile,...
    'houdiniproductionServerArchive\houdini.ctf',...
    'OutputDir','D:\Documents\MATLAB\work\HoudiniMPSAddIn',...
    'GenerateVisualBasicFile','on')
```

```
opts =

  ExcelClientForProductionServerOptions with properties:

                       AddInName: 'houdini'
                    AddInVersion: '1.0.0.0'
                       ClassName: 'houdiniClass'
                      DebugBuild: off
                   FunctionFiles: {'C:\Program Files\MATLAB\R2023a\extern\examples\compiler\hou
            GenerateVisualBasicFile: on
                     ServerArchive: 'houdiniproductionServerArchive\houdini.ctf'
          ReplaceExcelBlankWithNaN: off
          ConvertExcelDateToString: off
           ReplaceNaNToZeroInExcel: off
    ConvertNumericOutToDateInExcel: off
                           Verbose: off
                         OutputDir: 'D:\Documents\MATLAB\work\HoudiniMPSAddIn'
```

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

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

```
opts =
```

```
  ExcelClientForProductionServerOptions with properties:

                        AddInName: 'houdini'
                     AddInVersion: '1.0.0.0'
                        ClassName: 'houdiniClass'
                       DebugBuild: off
                    FunctionFiles: {'C:\Program Files\MATLAB\R2023a\extern\examples\compiler\hou
           GenerateVisualBasicFile: on
                    ServerArchive: 'houdiniproductionServerArchive\houdini.ctf'
          ReplaceExcelBlankWithNaN: off
          ConvertExcelDateToString: off
           ReplaceNaNToZeroInExcel: off
    ConvertNumericOutToDateInExcel: off
                           Verbose: on
                         OutputDir: 'D:\Documents\MATLAB\work\HoudiniMPSAddIn'
```

Use the `ExcelClientForProductionServerOptions` object as an input to the `compiler.build.excelClientForProductionServer` function to build an Excel add-in for MATLAB Production Server.

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

Input Arguments

`FunctionFiles` — MATLAB function files

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

List of files implementing MATLAB functions, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Files must have a `.m` extension.

Example: `{'myFunction1.m','myFunction2.m'}`

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

`Results` — Build results object

`Results` object

Build results, specified as a `compiler.build.Results` object. Create the `Results` object by saving the output from the `compiler.build.productionServerArchive` function.

`ServerArchive` — Excel add-in build options

character vector | string scalar

MATLAB Production Server archive deployed on the Production Server, specified as a character vector or a string scalar.

Example: `'mpsArchive.ctf'`

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: `'Verbose','on'`

`AddInName` — Name of Excel add-in
character vector | string scalar

Name of the Excel add-in, specified as a character vector or string scalar. The default name of the generated add-in is the first entry of the `FunctionFiles` argument. The name must begin with a letter and contain only alphabetic characters and underscores.

Example: `'AddInName','myAddIn'`

Data Types: `char` | `string`

`AddInVersion` — Add-in version
`'1.0.0.0'` (default) | character vector | string scalar

Add-in version, specified as a character vector or a string scalar.

Example: `'AddInVersion','4.0'`

Data Types: `char` | `string`

`ClassName` — Name of class
character vector | string scalar

Name of the generated class, specified as a character vector or a string scalar. You cannot specify this option if you use a `ClassMap` input. Class names must meet the Excel class name requirements.

The default value is the `AddInName` argument appended with `Class`.

Example: `'ClassName','MagicSquareClass'`

Data Types: `char` | `string`

`ConvertExcelDateToString` — Flag to convert date to string
`'off'` (default) | on/off logical value

Flag to convert Excel date to string, 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 compiler converts the Excel date datatype to MATLAB string.
- If you set this property to `'off'`, then dates are not converted.

Example: `'ConvertExcelDateToString','On'`

Data Types: `logical`

`ConvertNumericOutToDateInExcel` — Flag to convert numeric data to Excel date
`'off'` (default) | on/off logical value

Flag to convert numeric data to Excel date, 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 compiler converts numeric data to the Excel date datatype.
- If you set this property to `'off'`, then numeric data is not converted.

Example: `'ConvertNumericOutToDateInExcel','On'`

Data Types: `logical`

`DebugBuild` — Flag to enable debug symbols
`'on'` (default) | on/off logical value

Flag to enable debug symbols, 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 add-in is compiled with debug symbols.
- If you set this property to `'off'`, then the add-in is not compiled with debug symbols.

Example: `'DebugSymbols','On'`

Data Types: `logical`

`GenerateVisualBasicFile` — Flag to generate Visual Basic file
`'off'` (default) | on/off logical value

Flag to generate a Visual Basic file (`.bas`) and an Excel add-in file (`.xla`), 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 function generates an Excel add-in XLA file and a Visual Basic BAS file containing the Microsoft Excel Formula Function interface to the add-in.
- If you set this property to `'off'`, then the function does not generate a Visual Basic file or an Excel add-in file.

Example: `'GenerateVisualBasicFile','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 add-in name appended with `excelAddIn`.

Example: `'OutputDir','D:\Documents\MATLAB\work\mymagicexcelAddIn'`

Data Types: `char` | `string`

`ReplaceExcelBlankWithNaN` — Flag to convert blank Excel cells to NaN
`'off'` (default) | on/off logical value

Flag to convert blank Excel cells to NaN, 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 compiler converts blank Excel cells to NaN in the compiled artifact.
- If you set this property to `'off'`, then blank Excel cells are not converted.

Example: `'ReplaceExcelBlankWithNaN','On'`

Data Types: `logical`

`ReplaceNaNToZeroInExcel` — Flag to convert NaN entries to zero
`'off'` (default) | on/off logical value

Flag to convert NaN entries to zero, 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 compiler converts NaN entries from the compiled artifact to zero in Excel.
- If you set this property to `'off'`, then NaN entries are not converted.

Example: `'ReplaceNaNToZeroInExcel','On'`

Data Types: `logical`

`Verbose` — Flag to control build verbosity
`'off'` (default) | on/off logical value

Flag to control 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','On'`

Data Types: `logical`

Output Arguments

`opts` — Excel add-in build options
`ExcelClientForProductionServerOptions` object

Excel add-in build options, returned as an `ExcelClientForProductionServerOptions` object.

Version History

Introduced in R2021b

See Also

`compiler.build.excelClientForProductionServer` | `mcc`

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.
- `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.
- `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” on page 1-12.

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” (MATLAB Production Server).

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 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” (MATLAB Production Server).

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

`productionServerCompiler`

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.

Creation

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

- Create a production server archive using `compiler.build.productionServerArchive` (example on page 7-34).
- Create a COM component using `compiler.build.comComponent` (example on page 7-34).
- Create a C shared library using `compiler.build.cSharedLibrary` (example on page 7-34).
- Create a C++ shared library using `compiler.build.cppSharedLibrary` (example on page 7-35).
- Create a .NET assembly using `compiler.build.dotNETAssembly` (example on page 7-35).
- Create a Java package using `compiler.build.javaPackage` (example on page 7-36).
- Create a Python package using `compiler.build.pythonPackage` (example on page 7-36).
- Create an Excel add-in for MATLAB Production Server using `compiler.build.excelClientForProductionServer` (example on page 7-37).

Properties

`BuildType` — Build type

`'productionServerArchive'` | `'comComponent'` | `'cSharedLibrary'` | `'cppSharedLibrary'` | `'dotNETAssembly'` | `'javaPackage'` | `'pythonPackage'` | `'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.productionServerArchive`	`'productionServerArchive'`
`compiler.build.comComponent`	`'comComponent'`

compiler.build Function	Build Type
compiler.build.cSharedLibrary	'cSharedLibrary'
compiler.build.cppSharedLibrary	'cppSharedLibrary'
compiler.build.dotNETAssembly	'dotNETAssembly'
compiler.build.javaPackage	'javaPackage'
compiler.build.pythonPackage	'pythonPackage'
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
'productionServerArchive'	1×1 cell array {'path\to*ArchiveName*.ctf'}
'comComponent'	2×1 cell array {'path\to*ComponentName_ComponentVersion*.dll'} {'path\to\GettingStarted.html'}
'cSharedLibrary'	4×1 cell array {'path\to*LibraryName*.h'} {'path\to*LibraryName*.dll'} {'path\to*LibraryName*.lib'} {'path\to\GettingStarted.html'}
'cppSharedLibrary'	2×1 or 4×1 cell array Using the matlab-data interface: {'path\to\v2\'} {'path\to\GettingStarted.html'} Using the mwArray interface: {'path\to*LibraryName*.h'} {'path\to*LibraryName*.dll'} {'path\to*LibraryName*.lib'} {'path\to\GettingStarted.html'}

Build Type	Files
'dotNETAssembly'	4×1 cell array {'path\to*AssemblyName*.dll'} {'path\to*AssemblyName*Native.dll'} {'path\to*AssemblyName*_overview.html'} {'path\to\GettingStarted.html'}
'javaPackage'	3×1 cell array {'path\to*PackageName*.jar'} {'path\to\doc\'} {'path\to\GettingStarted.html'}
'pythonPackage'	3×1 cell array {'path\to\example\'} {'path\to\setup.py'} {'path\to\GettingStarted.html'}

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 | COMComponentOptions | CSharedLibraryOptions | CppSharedLibraryOptions | DotNETAssemblyOptions | JavaPackageOptions | PythonPackageOptions | 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
'productionServerArchive'	ProductionServerArchiveOptions
'comComponent'	COMComponentOptions
'cSharedLibrary'	CSharedLibraryOptions
'cppSharedLibrary'	CppSharedLibraryOptions
'dotNETAssembly'	DotNETAssemblyOptions
'javaPackage'	JavaPackageOptions
'pythonPackage'	PythonPackageOptions
'excelClientForProductionServer'	ExcelClientForProductionServerOptions

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 COM Component

Create a COM component on a Windows system and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

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

```
results =

  Results with properties:

              BuildType: 'comComponent'
                  Files: {2×1 cell}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.COMComponentOptions]
```

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

- `magicsquare_1_0.dll`
- `GettingStarted.html`

Get Build Information from C Library

Create a C library and save information about the build type, compiled files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

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

```
results =
```

```
  Results with properties:

             BuildType: 'cSharedLibrary'
                 Files: {4×1 cell}
IncludedSupportPackages: {}
               Options: [1×1 compiler.build.CSharedLibraryOptions]
```

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

- `magicsquare.dll`
- `magicsquare.lib`
- `magicsquare.h`
- `GettingStarted.html`

Get Build Information from C++ Library

Create a C++ library and save information about the build type, compiled files, support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

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

```
results =

  Results with properties:

             BuildType: 'cppSharedLibrary'
                 Files: {2×1 cell}
IncludedSupportPackages: {}
               Options: [1×1 compiler.build.CppSharedLibraryOptions]
```

The `Files` property contains the paths to the `v2` folder and `GettingStarted.html`.

Get Build Information from .NET Assembly

Create a .NET assembly on a Windows system and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

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

```
results =

  Results with properties:

             BuildType: 'dotNETAssembly'
                 Files: {4×1 cell}
IncludedSupportPackages: {}
               Options: [1×1 compiler.build.DotNETAssemblyOptions]
```

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

- `magicsquare.dll`
- `magicsquareNative.dll`
- `magicsquare_overview.dll`
- `GettingStarted.html`

Get Build Information from Java Package

Create a Java package and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

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

```
results =

  Results with properties:

              BuildType: 'javaPackage'
                  Files: {3×1 cell}
IncludedSupportPackages: {}
                Options: [1×1 compiler.build.JavaPackageOptions]
```

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

- `doc` folder
- `magicsquare.jar`
- `GettingStarted.html`

Get Build Information from Python Package

Create a Python package and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m`.

```
results = compiler.build.pythonPackage('magicsquare.m');
```

```
results =

  Results with properties:

            BuildType: 'pythonPackage'
                Files: {3×1 cell}
IncludedSupportPackages: {}
              Options: [1×1 compiler.build.PythonPackageOptions]
```

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

- `example` folder
- `setup.py`

- `GettingStarted.html`

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.comComponent` | `compiler.build.cSharedLibrary` | `compiler.build.cppSharedLibrary` | `compiler.build.dotNETAssembly` | `compiler.build.javaPackage` | `compiler.build.pythonPackage` | `compiler.build.excelClientForProductionServer`

compiler.package.microserviceDockerImage

Create a microservice Docker image using files generated by MATLAB Compiler SDK

Syntax

```
compiler.package.microserviceDockerImage(results)
compiler.package.microserviceDockerImage(results,Name,Value)
compiler.package.microserviceDockerImage(results,'Options',opts)
compiler.package.microserviceDockerImage(files,filepath,'ImageName',
imageName)
compiler.package.microserviceDockerImage(files,filepath,'ImageName',
imageName,Name,Value)
compiler.package.microserviceDockerImage(files,filepath,'Options',opts)
```

Description

`compiler.package.microserviceDockerImage(results)` creates a Docker image for files generated by the MATLAB Compiler SDK using the `compiler.build.Results` object `results`. The results object is created by the `compiler.build.productionServerArchive` function.

`compiler.package.microserviceDockerImage(results,Name,Value)` creates a Docker image using the `compiler.build.Results` object `results` and additional options specified as one or more name-value arguments. Options include the build folder, entry point command, and image name.

`compiler.package.microserviceDockerImage(results,'Options',opts)` creates a Docker image using the `compiler.build.Results` object `results` and additional options specified by a `MicroserviceDockerImageOptions` object `opts`. If you use a `MicroserviceDockerImageOptions` object, you cannot specify any other options using name-value arguments.

`compiler.package.microserviceDockerImage(files,filepath,'ImageName', imageName)` creates a Docker image using `files` that are generated by MATLAB Compiler SDK. The Docker image name is specified by `imageName`.

`compiler.package.microserviceDockerImage(files,filepath,'ImageName', imageName,Name,Value)` creates a Docker image using `files` that are generated by MATLAB Compiler SDK. Additional options are specified as one or more name-value arguments.

`compiler.package.microserviceDockerImage(files,filepath,'Options',opts)` creates a Docker image using `files` that are generated by MATLAB Compiler SDK and additional options specified by a `MicroserviceDockerImageOptions` object `opts`. If you use a `MicroserviceDockerImageOptions` object, you cannot specify any other options using name-value arguments.

Examples

Create Microservice Docker Image Using Results

Create a microservice Docker image from a production server archive.

Install and configure Docker on your system. For details, see the prerequisites section of “Create Microservice Docker Image” on page 1-12.

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.

To deploy the image to Docker using port 9900 on the host machine, run the following command in a system terminal:

```
docker run --rm -p 9900:9910 magicsquare
```

Customize Microservice Docker Image Using Results and Name Value Arguments

Customize a microservice image using name-value arguments on a Linux system to specify the image name and build directory.

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);
```

Call the `compiler.package.microserviceDockerImage` function using the `Results` object. Use name-value pair arguments to specify the image name and build folder, and disable the call to build the Docker image.

```
compiler.package.microserviceDockerImage(buildResults,...
'ImageName','mymagicapp',...
'DockerContext','/home/mluser/Documents/MATLAB/docker',...
'ExecuteDockerBuild','Off');
```

This syntax populates the context folder with the Docker files.

After you have examined the generated files, use the command `docker build` to build the Docker image. For details, refer to the Docker documentation.

Customize Microservice Docker Image Using Results and Options Object

Customize a Docker image using a `MicroserviceDockerImageOptions` object.

Write a function named `hello-world.m` using the following code.

```
disp('Hello world!');
```

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

```
buildResults = compiler.build.productionServerArchive('hello-world.m');
```

Create a `MicroserviceDockerImageOptions` object to specify additional build options.

```
opts = compiler.package.microserviceDockerImageOptions(buildResults,
'DockerContext','hellodocker')
```

```
opts =

  MicroserviceDockerImageOptions with properties:

    ExecuteDockerBuild: on
             ImageName: 'helloworld'
         DockerContext: 'hellodocker'
```

Pass the `MicroserviceDockerImageOptions` and `Results` objects as inputs to the `compiler.package.microserviceDockerImage` function to build the Docker image.

```
compiler.package.microserviceDockerImage(buildResults,'Options',opts);
```

Create Microservice Docker Image Using Files and Name Value Arguments

Create a Docker image using files generated by MATLAB Compiler SDK and specify the image name.

Build a production server archive using the `mcc` command.

```
mcc -W CTF:myapp -U magicsquare.m
```

Build the Docker image by passing the generated files to the `compiler.package.microserviceDockerImage` function.

```
compiler.package.microserviceDockerImage('myapp.ctf',...
'requiredMCRProducts.txt','ImageName','microapp');
```

Customize Microservice Docker Image Using Files and Options Object

Customize a Docker image using files generated by MATLAB Compiler SDK and a `MicroserviceDockerImageOptions` object.

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

```
buildResults = compiler.build.productionServerArchive('helloworld.m');
```

Create a `MicroserviceDockerImageOptions` object to specify additional build options, such as the build folder.

```
opts = compiler.package.MicroserviceDockerImageOptions(buildResults,...
'DockerContext','DockerImages')
```

You can modify property values of an existing `MicroserviceDockerImageOptions` object using dot notation. For example, disable the call to build the Docker image.

```
opts.ExecuteDockerBuild = 'Off';
```

Populate the `DockerContext` folder with the Docker files by passing the files and options object to the `compiler.package.microserviceDockerImage` function.

```
cd helloworldproductionServerArchive

compiler.package.microserviceDockerImage('helloworld',...
'requiredMCRProducts.txt','Options',opts);
```

Input Arguments

`results` — Build results

`compiler.build.Results` object

Build results created by the `compiler.build.productionServerArchive` function, specified as a `compiler.build.Results` object.

`files` — Files and folders for installation

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

Files and folders for installation, specified as a character vector, string scalar, string array, or cell array of strings. Exactly one of these files must be a CTF file generated by MATLAB Compiler SDK. The list can also include any additional files and folders required by the service to run. You can package files generated by MATLAB Compiler SDK in a particular release using the `compiler.package.microserviceDockerImage` function of the same release.

Example: `'myDockerFiles/'`

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

`filepath` — Path to `requiredMCRProducts.txt` file

character vector | string scalar

Path to the `requiredMCRProducts.txt` file, specified as a character vector or string scalar. This file is generated by MATLAB Compiler SDK. The path can be relative to the current working directory or absolute.

Example: `'/home/mluser/Documents/MATLAB/magicsquare/requiredMCRProducts.txt'`

Data Types: `char` | `string`

`imageName` — Name of Docker image

character vector | string scalar

Name of the Docker image. It must comply with Docker naming rules.

Example: `'hello-world'`

Data Types: `char` | `string`

`opts` — Docker options
`MicroserviceDockerImageOptions` object

Microservice Docker options, specified as a `MicroserviceDockerImageOptions` object.

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.

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

Example: `'ExecuteDockerBuild','on'`

`AdditionalCommands` — Additional commands to pass to Docker image
`''` (default) | character vector | string scalar | cell array of character vectors

Additional commands to pass to the Docker image, specified as a character vector, a string scalar, or a cell array of character vectors. Commands are added to the `Dockerfile` and execute during image generation.

Example: `'AdditionalCommands','top'`

Data Types: `char` | `string`

`AdditionalPackages` — Additional packages to install into Docker image
`''` (default) | character vector | string scalar | cell array of character vectors

Additional Ubuntu® 20.04 packages to install into the Docker image, specified as a character vector, a string scalar, or a cell array of character vectors.

Example: `'AdditionalPackages,'syslog-ng'`

Data Types: `char` | `string`

`DockerContext` — Path to build folder
`'`*`ImageName`*`docker'` (default) | character vector | string scalar

Path to the build folder where the Docker image is built, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

If no path is specified, the function creates a build folder named *`ImageName`*`docker` in the current working directory.

Example: `'DockerContext','/home/mluser/Documents/MATLAB/docker/magicsquaredocker'`

Data Types: `char` | `string`

`ExecuteDockerBuild` — Flag to build Docker image
`'on'` (default) | on/off logical value

Flag to build the Docker image, 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 function will build the Docker image.
- If you set this property to `'off'`, then the function will populate the `DockerContext` folder without calling 'docker build'.

Example: `'ExecuteDockerBuild','Off'`

Data Types: `logical`

`ImageName` — Name of Docker image
`''` (default) | character vector | string scalar

Name of the Docker image, specified as a character vector or a string scalar. The name must comply with Docker naming rules. Docker repository names must be lowercase. If the main executable or archive file is named using uppercase letters, then the uppercase letters are replaced with lowercase letters in the Docker image name.

Example: `'ImageName','magicsquare'`

Data Types: `char` | `string`

Limitations

- In R2022a, this function is only supported on Linux operating systems.

Version History

Introduced in R2022a

See Also

`compiler.package.MicroserviceDockerImageOptions` | `compiler.build.Results` | `compiler.build.productionServerArchive`

Topics

“Create Microservice Docker Image” on page 1-12

compiler.package.MicroserviceDockerImageOptions

Create a microservice Docker options object

Syntax

```
opts = compiler.package.MicroserviceDockerImageOptions(results)
opts = compiler.package.MicroserviceDockerImageOptions(results,Name,Value)
opts = compiler.package.MicroserviceDockerImageOptions('ImageName',imageName)
opts = compiler.package.MicroserviceDockerImageOptions('ImageName',imageName,
Name,Value)
```

Description

`opts = compiler.package.MicroserviceDockerImageOptions(results)` creates a `MicroserviceDockerImageOptions` object `opts` using the `compiler.build.Results` object `results`. The `Results` object is created by the `compiler.build.productionServerArchive` function. Pass the `MicroserviceDockerImageOptions` object as an input to the `compiler.package.docker` function to specify build options.

`opts = compiler.package.MicroserviceDockerImageOptions(results,Name,Value)` creates a `MicroserviceDockerImageOptions` object `opts` using the `compiler.build.Results` object `results` and additional options specified as one or more pairs of name-value arguments. Options include the build folder, entry point command, and image name.

`opts = compiler.package.MicroserviceDockerImageOptions('ImageName',imageName)` creates a default `MicroserviceDockerImageOptions` object with the image name specified by `imageName`.

`opts = compiler.package.MicroserviceDockerImageOptions('ImageName',imageName, Name,Value)` creates a generic `MicroserviceDockerImageOptions` object with the image name specified by `imageName` and additional options specified as one or more pairs of name-value arguments.

Examples

Create Microservices Docker Options Object Using Build Results

Create a `MicroserviceDockerImageOptions` object using the build results from a production server archive.

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);
```

Create a `MicroserviceDockerImageOptions` object using the build results from the `compiler.build.productionServerArchive` function.

```
opts = compiler.package.MicroserviceDockerImageOptions(buildResults);
```

You can modify property values of an existing `MicroserviceDockerImageOptions` object using dot notation. For example, set the build folder.

```
opts.DockerContext = 'myDockerFiles';
```

Pass the `MicroserviceDockerImageOptions` and `Results` objects as inputs to the `compiler.package.microserviceDockerImage` function to build the microservice Docker image.

```
compiler.package.microserviceDockerImage(buildResults,'Options',opts);
```

Customize Microservice Docker Options Object Using Build Results

Create a `MicroserviceDockerImageOptions` object using build results from a production server archive and customize it using name-value arguments.

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);
```

Create a `MicroserviceDockerImageOptions` object using the build results from the `compiler.build.productionServerArchive` function. Use name-value arguments to specify the image name and build folder.

```
opts = compiler.package.MicroserviceDockerImageOptions(buildResults,...
'DockerContext','Docker/MagicSquareMicroservice',...
'ImageName','magicsquare-microservice-')
```

```
opts =

  MicroserviceDockerImageOptions with properties:

    AdditionalCommands: {}
        AdditionalPackages: {}
        ExecuteDockerBuild: on
                 ImageName: 'magic-square-'
             DockerContext: './Docker/MagicSquareMicroservice/magicsquare-microservice-docker'
```

Create Microservices Docker Options Object Using Image Name

Create a generic `MicroserviceDockerImageOptions` object by only specifying the image name.

Create a `MicroserviceDockerImageOptions` object.

```
opts = compiler.package.MicroserviceDockerImageOptions('ImageName','helloworld')
```

```
opts =

  MicroserviceDockerImageOptions with properties:

    AdditionalCommands: {}
        AdditionalPackages: {}
        ExecuteDockerBuild: on
```

```
                ImageName: 'helloworld'
            DockerContext: './helloworlddocker'
```

Customize Microservices Docker Options Object Using Image Name

Create a `MicroserviceDockerImageOptions` object using the image name and customize it using name-value arguments.

Create a `MicroserviceDockerImageOptions` object. Use name-value arguments to specify the image name and build folder.

```
opts = compiler.package.MicroserviceDockerImageOptions('ImageName','myapp',...
'DockerContext','Docker/MyDockerApp')

opts =

  MicroserviceDockerImageOptions with properties:

    AdditionalCommands: {}
        AdditionalPackages: {}
        ExecuteDockerBuild: on
                 ImageName: 'myapp'
             DockerContext: './Docker/MyDockerApp'
```

Input Arguments

`results` — Build results

`compiler.build.Results` object

Build results from the `compiler.build.productionServerArchive` function, specified as a `compiler.build.Results` object.

`imageName` — Name of Docker image

character vector | string scalar

Name of the Docker image. It must comply with Docker naming rules.

Example: `'hello-world'`

Data Types: `char` | `string`

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.

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

Example: `'ExecuteDockerBuild','on'`

`AdditionalCommands` — Additional commands to pass to Docker image

`''` (default) | character vector | string scalar | cell array of character vectors

Additional commands to pass to the Docker image, specified as a character vector, a string scalar, or a cell array of character vectors. Commands are added to the `Dockerfile` and execute during image generation.

Example: `'AdditionalCommands','top'`

Data Types: `char` | `string`

`AdditionalPackages` — Additional packages to install into Docker image
`''` (default) | character vector | string scalar | cell array of character vectors

Additional Ubuntu 20.04 packages to install into the Docker image, specified as a character vector, a string scalar, or a cell array of character vectors.

Example: `'AdditionalPackages,'syslog-ng'`

Data Types: `char` | `string`

`DockerContext` — Path to build folder
`'`*`ImageName`*`docker'` (default) | character vector | string scalar

Path to the build folder where the Docker image is built, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

If no path is specified, the function creates a build folder named *`ImageName`*`docker` in the current working directory.

Example: `'DockerContext','/home/mluser/Documents/MATLAB/docker/magicsquaredocker'`

Data Types: `char` | `string`

`ExecuteDockerBuild` — Flag to build Docker image
`'on'` (default) | on/off logical value

Flag to build the Docker image, 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 function will build the Docker image.
- If you set this property to `'off'`, then the function will populate the `DockerContext` folder without calling 'docker build'.

Example: `'ExecuteDockerBuild','Off'`

Data Types: `logical`

`ImageName` — Name of Docker image
`''` (default) | character vector | string scalar

Name of the Docker image, specified as a character vector or a string scalar. The name must comply with Docker naming rules. Docker repository names must be lowercase. If the main executable or archive file is named using uppercase letters, then the uppercase letters are replaced with lowercase letters in the Docker image name.

Example: `'ImageName','magicsquare'`

Data Types: `char` | `string`

Output Arguments

`opts` — Microservice Docker options object
`MicroserviceDockerImageOptions` object

Microservice Docker image build options, returned as a `MicroserviceDockerImageOptions` object.

Limitations

- In R2022a, this function is only supported on Linux operating systems.

Version History

Introduced in R2022a

See Also

`compiler.package.microserviceDockerImage` | `compiler.build.Results` | `compiler.build.productionServerArchive`

compiler.runtime.createInstallerDockerImage

Create a MATLAB Runtime installer Docker image on offline machines

Syntax

```
compiler.runtime.createInstallerDockerImage()
compiler.runtime.createInstallerDockerImage(filepath)
```

Description

Note You do not need to run this command if you are connected to the Docker image repository.

`compiler.runtime.createInstallerDockerImage()` creates a MATLAB Runtime installer Docker image using the installer file provided by the `compiler.runtime.installer` function, in cases where MATLAB is unable to reach the Docker image repository. The installer image is used to create microservice Docker images using `compiler.package.docker` and `compiler.package.microserviceDockerImage`. This workflow is only supported on Linux.

`compiler.runtime.createInstallerDockerImage(filepath)` creates a MATLAB Runtime installer Docker image using the installer file provided by `filepath`. This workflow is supported on all platforms.

Examples

Build Runtime Installer Docker Image on Linux

Here, you create a MATLAB Runtime installer Docker image on Linux.

Install and configure Docker on your system.

Create the Docker image.

```
compiler.runtime.createInstallerDockerImage()
```

Build Runtime Installer Docker Image on Other Platforms

Here, you create a MATLAB Runtime installer Docker image on Windows for R2023a.

Install and configure Docker on your system. For details, see the prerequisites section of “Create Microservice Docker Image” on page 1-12.

Download the MATLAB Runtime installer for Linux for the R2023a release from https://www.mathworks.com/products/compiler/matlab-runtime.html.

Create the Docker image using the path to the installer archive. For example, if it is located in the `Downloads` folder of `mwuser`, type the following command.

```
compiler.runtime.createInstallerDockerImage("C:\Users\mwuser\Downloads\MATLAB_Runtime_R2023a_glnx
```

Input Arguments

`filepath` — Path to MATLAB Runtime installer file for Linux

character vector | string scalar

Path to the MATLAB Runtime installer file for Linux, specified as a character vector or string scalar. The path can be relative to the current working directory or absolute.

Example: `"C:\Users\mwuser\Downloads\MATLAB_Runtime_R2022b_Update_1_glnxa64.zip"`

Data Types: `char` | `string`

Version History

Introduced in R2022b

See Also

`compiler.package.docker` | `compiler.package.microserviceDockerImage` | `compiler.runtime.download`

Topics

“Package MATLAB Standalone Applications into Docker Images”
“Create Microservice Docker Image” on page 1-12

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.

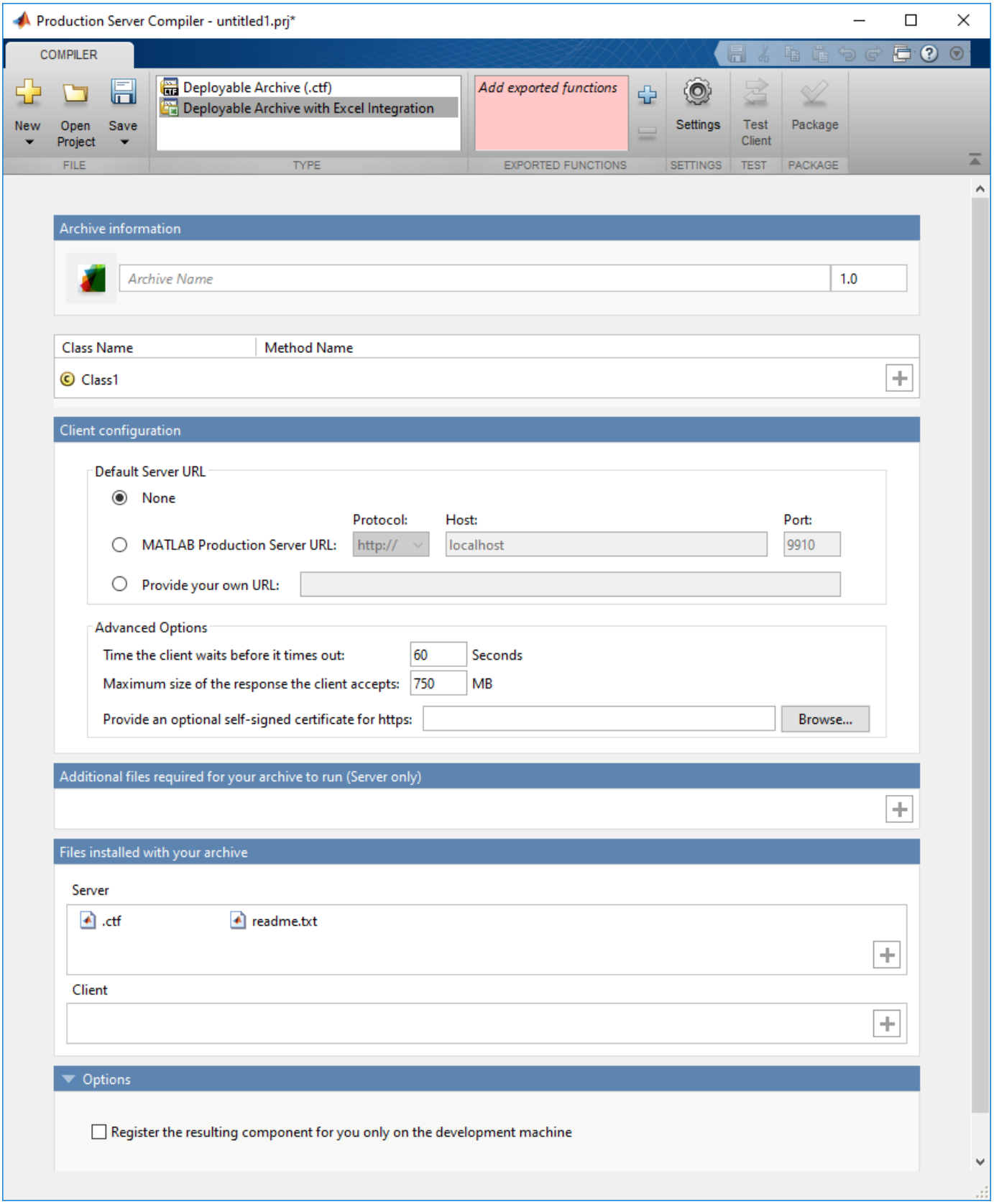
Apps

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.


Production Server Compiler - untitled1.prj*
COMPILER
New
Open Project
Save
FILE
Deployable Archive (.ctf)
Deployable Archive with Excel Integration
TYPE
Add exported functions
EXPORTED FUNCTIONS
Settings
SETTINGS
Test Client
TEST
Package
PACKAGE
Archive information
Archive Name
1.0
Class Name
Method Name
Class1
Client configuration
Default Server URL
None
Protocol:
Host:
Port:
MATLAB Production Server URL:
http://
localhost
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” on page 4-3

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” on page 4-3

Client Programming

Create MATLAB Production Server Java Client Using MWHttpClient Class

This example shows how to write a MATLAB Production Server client using the `MWHttpClient` class from the Java client API. For information on obtaining the Java client library, see “Obtain and Configure Client Library” (MATLAB Production Server). In your Java code, you will:

- Define a Java interface that represents the deployed MATLAB function.
- Instantiate a static proxy object to communicate with the server.
- Call the deployed function in your Java code.

To create a Java MATLAB Production Server client application:

1. Create a new file, for example, `MPSClientExample.java`.
2. Using a text editor, open `MPSClientExample.java`.
3. Add the following import statements to the file:

```
import java.net.URL;
import java.io.IOException;
import com.mathworks.mps.client.MWClient;
import com.mathworks.mps.client.MWHttpClient;
import com.mathworks.mps.client.MATLABException;
```

4. Add a Java interface that represents the deployed MATLAB function.

   For example, consider the following `addmatrix` function deployed to the server. For information on writing and compiling the function for deployment, see “Create Deployable Archive for MATLAB Production Server” (MATLAB Production Server). For deploying the function to the server, see “Deploy Archive to MATLAB Production Server” (MATLAB Production Server).

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

a = a1 + a2;
```

   The interface for the `addmatrix` function follows.

```
interface MATLABAddMatrix {
       double[][] addmatrix(double[][] a1, double[][] a2)
         throws MATLABException, IOException;
   }
```

   When creating the interface, note the following:

   - You can give the interface any valid Java name.
   - You must give the method defined by this interface the same name as the deployed MATLAB function.
   - The Java method must support the same inputs and outputs supported by the MATLAB function, in both type and number. For more information about data type conversions and how to handle more complex MATLAB function signatures, see “Data Conversion with Java and MATLAB Types” (MATLAB Production Server) and “Conversion of Java Types to MATLAB Types” (MATLAB Production Server).
   - The Java method must handle MATLAB exceptions and I/O exceptions.

5. Add the following class definition:

```
public class MPSClientExample
{
}
```

This class now has a single `main` method that calls the generated class.

6 Add the `main()` method to the application.

```
public static void main(String[] args)
{
}
```

7 Add the following code to the top of the `main()` method to initialize the variables used by the application:

```
double[][] a1={{1,2,3},{3,2,1}};
double[][] a2={{4,5,6},{6,5,4}};
```

8 Instantiate a client object using the `MWHttpClient` constructor.

```
MWClient client = new MWHttpClient();
```

This class establishes an HTTP connection between the application and the server instance.

9 Call the `createProxy` method of the client object to create a dynamic proxy.

You must specify the URL of the deployable archive and the name of your interface `class` as arguments:

```
MATLABAddMatrix m = client.createProxy(new URL("http://localhost:9910/addmatrix"),
                                       MATLABAddMatrix.class);
```

The URL value (`"http://localhost:9910/addmatrix"`) used to create the proxy contains three parts:

- the server address (`localhost`).
- the port number (`9910`).
- the archive name (`addmatrix`)

For more information about the `createProxy` method, see the Javadoc included in the *`matlabroot`*`/toolbox/compiler_sdk/mps_clients` folder.

10 Call the deployed MATLAB function in your Java application by calling the public method of the interface.

```
  double[][] result = m.addmatrix(a1,a2);
```

11 Call the `close()` method of the client object to free system resources.

```
client.close();
```

12 Save the Java file.

The completed Java file should resemble the following:

```
import java.net.URL;
import java.io.IOException;
import com.mathworks.mps.client.MWClient;
import com.mathworks.mps.client.MWHttpClient;
import com.mathworks.mps.client.MATLABException;

interface MATLABAddMatrix
  {
    double[][] addmatrix(double[][] a1, double[][] a2)
         throws MATLABException, IOException;
  }

public class MPSClientExample {
```

```
    public static void main(String[] args){

        double[][] a1={{1,2,3},{3,2,1}};
        double[][] a2={{4,5,6},{6,5,4}};

        MWClient client = new MWHttpClient();

        try{
            MATLABAddMatrix m = client.createProxy(new URL("http://localhost:9910/addmatrix"),
                                                   MATLABAddMatrix.class);
            double[][] result = m.addmatrix(a1,a2);

            // Print the resulting matrix
            printResult(result);

        }catch(MATLABException ex){

            // This exception represents errors in MATLAB
               System.out.println(ex);
        }catch(IOException ex){

            // This exception represents network issues.
               System.out.println(ex);
        }finally{

            client.close();
        }
    }

    private static void printResult(double[][] result){
        for(double[] row : result){
            for(double element : row){
                System.out.print(element + " ");
            }
            System.out.println();
        }
    }
}
```

13 Compile the Java application, using the `javac` command or use the build capability of your Java IDE.

For example, enter the following at the Windows command prompt:

```
javac -classpath "matlabroot\toolbox\compiler_sdk\mps_clients\java\mps_client.jar" MPSClientExample.java
```

14 Run the application using the `java` command or your IDE.

For example, enter the following at the Windows command prompt:

```
java -classpath .;"matlabroot\toolbox\compiler_sdk\mps_clients\java\mps_client.jar" MPSClientExample
```

To run the application on Linux and macOS systems, use a colon (`:`) to separate multiple paths.

The application returns the following at the console:

```
5.0 7.0 9.0
9.0 7.0 5.0
```

See Also

More About

- “Bond Pricing Tool for Java Client” (MATLAB Production Server)
- “MATLAB Production Server Java Client Basics” (MATLAB Production Server)
- “Synchronous RESTful Requests Using Protocol Buffers in the Java Client” (MATLAB Production Server)

- “Asynchronous RESTful Requests Using Protocol Buffers in the Java Client” (MATLAB Production Server)

Create a C# Client

This example shows how to write a C# application to call a MATLAB function deployed to MATLAB Production Server. The C# application uses the MATLAB Production Server .NET client library.

A .NET application programmer typically performs this task. The tutorial assumes that you have Microsoft Visual Studio® and .NET installed on your computer.

Create Microsoft Visual Studio Project

1. Open Microsoft Visual Studio.
2. Click **File > New > Project**.
3. In the New Project dialog box, select the template you want to use. For example, if you want to create a C# console application in Visual Studio 2017, select **Visual C# > Windows Desktop** in the left navigation pane, then select the **Console App (.Net Framework)**.
4. Type the name of the project in the **Name** field (for example, `Magic`).
5. Click **OK**. Your `Magic` source shell is created, typically named `Program.cs`, by default.

Create Reference to Client Runtime Library

Create a reference in your `Magic` project to the MATLAB Production Server client runtime library. In Microsoft Visual Studio, perform the following steps:

1. In the **Solution Explorer** pane within Microsoft Visual Studio (usually on the right side), right-click your `Magic` project, select **Add > Browse**.
2. Browse to the MATLAB Production Server .NET client runtime library location.

   The library is located in *`matlabroot`*`\toolbox\compiler_sdk\mps_clients\dotnet`. Select the `MathWorks.MATLAB.ProductionServer.Client.dll` file.

   The client library is also available for download at `https://www.mathworks.com/products/matlab-production-server/client-libraries.html`.
3. Click **OK**. Your Microsoft Visual Studio project now references the `MathWorks.MATLAB.ProductionServer.Client.dll`.

Deploy MATLAB Function to Server

Write a MATLAB function `mymagic` that uses the `magic` function to create a magic square, package `mymagic` into a deployable archive called `mymagic_deployed`, then deploy it to a server. The function `mymagic` takes a single `int` input and returns a magic square as a 2-D `double` array. The example assumes that the server instance is running at `http://localhost:9910`.

```
function m = mymagic(in)
    m = magic(in);
```

Design .NET Interface in C#

Invoke the deployed MATLAB function `mymagic` from a .NET client through a .NET interface. Design a C# interface `Magic` to match the MATLAB function `mymagic`.

- The .NET interface has the same number of inputs and outputs as the MATLAB function.
- Since you are deploying one MATLAB function on the server, you define one corresponding .NET method in your C# code.

- Both the MATLAB function and the .NET interface process the same data types—input type `int` and output type 2-D `double`.
- In your C# client program, use the interface `Magic` to specify the type of the proxy object reference in the `CreateProxy` method. The `CreateProxy` method requires the URL to the deployable archive that contains the `mymagic` function (`http://localhost:9910/mymagic_deployed`) as an input argument.

```
public interface Magic
       {
         double[,] mymagic(int in1);
       }
```

Write, Build, and Run .NET Application

1. Open the Microsoft Visual Studio project `Magic` that you created earlier.
2. In the `Program.cs` tab, paste in the code below.

```
using System;
using System.Net;
using MathWorks.MATLAB.ProductionServer.Client;

namespace Magic
{
    public class MagicClass
    {

        public interface Magic
        {
            double[,] mymagic(int in1);
        }

        public static void Main(string[] args)
        {
            MWClient client = new MWHttpClient();
            try
            {
                Magic me = client.CreateProxy<Magic>
                          (new Uri("http://localhost:9910/mymagic_deployed"));
                double[,] result1 = me.mymagic(4);
                print(result1);
            }
            catch (MATLABException ex)
            {
                Console.WriteLine("{0} MATLAB exception caught.", ex);
                Console.WriteLine(ex.StackTrace);
            }
            catch (WebException ex)
            {
                Console.WriteLine("{0} Web exception caught.", ex);
                Console.WriteLine(ex.StackTrace);
            }
            finally
            {
                client.Dispose();
            }
            Console.ReadLine();
        }

        public static void print(double[,] x)
        {
            int rank = x.Rank;
            int[] dims = new int[rank];

            for (int i = 0; i < rank; i++)
            {
                dims[i] = x.GetLength(i);
            }
```

```
            for (int j = 0; j < dims[0]; j++)
            {
                for (int k = 0; k < dims[1]; k++)
                {
                    Console.Write(x[j, k]);
                    if (k < (dims[1] - 1))
                    {
                        Console.Write(",");
                    }
                }
                Console.WriteLine();
            }
        }
    }
}
```

The URL value (`"http://localhost:9910/mymagic_deployed"`) used to create the proxy contains three parts.

- the server address (`localhost`).
- the port number (`9910`).
- the archive name (`mymagic_deployed`).

3. Build the application. Click **Build** > **Build Solution**.
4. Run the application. Click **Debug** > **Start Without Debugging**. The program returns the following console output.

```
16,2,3,13
5,11,10,8
9,7,6,12
4,14,15,1
```

See Also

More About

- “Create a .NET MATLAB Production Server Client” (MATLAB Production Server)
- “Configure the Client-Server Connection” (MATLAB Production Server)
- “Synchronous RESTful Requests Using Protocol Buffers in .NET Client” (MATLAB Production Server)

Create a Python Client

This example shows how to write a MATLAB Production Server client using the Python client API. The client application calls the `addmatrix` MATLAB function deployed to a server instance. For information on writing and compiling the function for deployment, see “Create Deployable Archive for MATLAB Production Server” (MATLAB Production Server). For deploying the function to the server, see “Deploy Archive to MATLAB Production Server” (MATLAB Production Server).

Before you write the client application, you must have the MATLAB Production Server Python client libraries installed on your system. For details, see “Install the MATLAB Production Server Python Client” (MATLAB Production Server).

1. Start the Python command line interpreter.
2. Enter the following import statements at the Python command prompt.

   ```
   import matlab
   from production_server import client
   ```
3. Open the connection to the MATLAB Production Server instance and initialize the client runtime.

   ```
   client_obj = client.MWHttpClient("http://localhost:9910")
   ```
4. Create the MATLAB data to input to the function.

   ```
   a1 = matlab.double([[1,2,3],[3,2,1]])
   a2 = matlab.double([[4,5,6],[6,5,4]])
   ```
5. Call the deployed MATLAB function. To call the function, you must know the name of the deployed archive and the name of the function.

   The syntax for invoking a function is `client.`*`archiveName`*`.`*`functionName`*`(`*`arg1`*`, `*`arg2`*`, .., [nargout=`*`numOutArgs`*`])`.

   ```
   client_obj.addmatrix.addmatrix(a1,a2)
   ```

   The output is:

   ```
   matlab.double([[5.0,7.0,9.0],[9.0,7.0,5.0]])
   ```
6. Close the client connection.

   ```
   client_obj.close()
   ```

See Also

`matlab.production_server.client.MWHttpClient`

Related Examples

- “Create Client Connection” (MATLAB Production Server)
- “Invoke Packaged MATLAB Functions” (MATLAB Production Server)

Create a C++ Client

This example shows how to write a MATLAB Production Server client using the C client API. The client application calls the `addmatrix` function you compiled in “Package Deployable Archives with Production Server Compiler App” and deployed in “Deploy Archive to MATLAB Production Server” (MATLAB Production Server).

Create a C++ MATLAB Production Server client application:

1. Create a file called `addmatrix_client.cpp`.
2. Using a text editor, open `addmatrix_client.cpp`.
3. Add the following include statements to the file:

```
#include <iostream>
#include <mps/client.h>
```

Note The header files for the MATLAB Production Server C client API are located in the *`matlabroot`*`/toolbox/compiler_sdk/mps_clients/c/include/mps` folder.

4. Add the `main()` method to the application.

```
int main ( void )
{
}
```

5. Initialize the client runtime.

```
mpsClientRuntime* mpsruntime = mpsInitializeEx(MPS_CLIENT_1_1);
```

6. Create the client configuration.

```
mpsClientConfig* config;
mpsStatus status = mpsruntime->createConfig(&config);
```

7. Create the client context.

```
mpsClientContext* context;
status = mpsruntime->createContext(&context, config);
```

8. Create the MATLAB data to input to the function.

```
double a1[2][3] = {{1,2,3},{3,2,1}};
double a2[2][3] = {{4,5,6},{6,5,4}};

int numIn=2;
mpsArray** inVal = new mpsArray* [numIn];

inVal[0] = mpsCreateDoubleMatrix(2,3,mpsREAL);
inVal[1] = mpsCreateDoubleMatrix(2,3,mpsREAL);

double* data1 = (double *)( mpsGetData(inVal[0]) );
double* data2 = (double *)( mpsGetData(inVal[1]) );

for(int i=0; i<2; i++)
{
  for(int j=0; j<3; j++)
  {
    mpsIndex subs[] = { i, j };
    mpsIndex id = mpsCalcSingleSubscript(inVal[0], 2, subs);
    data1[id] = a1[i][j];
    data2[id] = a2[i][j];
```

```
  }
}
```

9 Create the MATLAB data to hold the output.

```
int numOut = 1;
mpsArray **outVal = new mpsArray* [numOut];
```

10 Call the deployed MATLAB function.

Specify the following as arguments:

- client context
- URL of the function
- Number of expected outputs
- Pointer to the `mpsArray` holding the outputs
- Number of inputs
- Pointer to the `mpsArray` holding the inputs

```
mpsStatus status = mpsruntime->feval(context,
    "http://localhost:9910/addmatrix/addmatrix",
    numOut, outVal, numIn, (const mpsArray**)inVal);
```

For more information about the `feval` function, see the reference material included in the *`matlabroot`*`/toolbox/compiler_sdk/mps_clients` folder.

11 Verify that the function call was successful using an `if` statement.

```
if (status==MPS_OK)
{
}
```

12 Inside the `if` statement, add code to process the output.

```
double* out = mpsGetPr(outVal[0]);

for (int i=0; i<2; i++)
{
  for (int j=0; j<3; j++)
  {
    mpsIndex subs[] = {i, j};
    mpsIndex id = mpsCalcSingleSubscript(outVal[0], 2, subs);
    std::cout << out[id] << "\t";
  }
  std::cout << std::endl;
}
```

13 Add an `else` clause to the `if` statement to process any errors.

```
else
{
  mpsErrorInfo error;
  mpsruntime->getLastErrorInfo(context, &error);
  std::cout << "Error: " << error.message << std::endl;
  switch(error.type)
  {
    case MPS_HTTP_ERROR_INFO:
      std::cout << "HTTP: " << error.details.http.responseCode << ": "
          << error.details.http.responseMessage << std::endl;
    case MPS_MATLAB_ERROR_INFO:
      std::cout << "MATLAB: " << error.details.matlab.identifier
```

```
            << std::endl;
        std::cout << error.details.matlab.message << std::endl;
      case MPS_GENERIC_ERROR_INFO:
        std::cout << "Generic: " << error.details.general.genericErrorMsg
            << std::endl;
    }

    mpsruntime->destroyLastErrorInfo(&error);
  }
```

14 Free the memory used by the inputs.

```
  for (int i=0; i<numIn; i++)
    mpsDestroyArray(inVal[i]);
  delete[] inVal;
```

15 Free the memory used by the outputs.

```
  for (int i=0; i<numOut; i++)
    mpsDestroyArray(outVal[i]);
  delete[] outVal;
```

16 Free the memory used by the client runtime.

```
  mpsruntime->destroyConfig(config);
  mpsruntime->destroyContext(context);
  mpsTerminate();
```

17 Save the file.

The completed program should resemble the following:

```
#include <iostream>
#include <mps/client.h>

int main ( void )
{
  mpsClientRuntime* mpsruntime = mpsInitializeEx(MPS_CLIENT_1_1);

  mpsClientConfig* config;
  mpsStatus status = mpsruntime->createConfig(&config);

  mpsClientContext* context;
  status = mpsruntime->createContext(&context, config);

  double a1[2][3] = {{1,2,3},{3,2,1}};
  double a2[2][3] = {{4,5,6},{6,5,4}};

  int numIn=2;
  mpsArray** inVal = new mpsArray* [numIn];
  inVal[0] = mpsCreateDoubleMatrix(2,3,mpsREAL);
  inVal[1] = mpsCreateDoubleMatrix(2,3,mpsREAL);
  double* data1 = (double *)( mpsGetData(inVal[0]) );
  double* data2 = (double *)( mpsGetData(inVal[1]) );
  for(int i=0; i<2; i++)
  {
    for(int j=0; j<3; j++)
    {
      mpsIndex subs[] = { i, j };
      mpsIndex id = mpsCalcSingleSubscript(inVal[0], 2, subs);
      data1[id] = a1[i][j];
      data2[id] = a2[i][j];
    }
  }

  int numOut = 1;
  mpsArray **outVal = new mpsArray* [numOut];

  status = mpsruntime->feval(context,
              "http://localhost:9910/addmatrix/addmatrix",
              numOut, outVal, numIn, (const mpsArray **)inVal);

  if (status==MPS_OK)
  {
    double* out = mpsGetPr(outVal[0]);
```

```
    for (int i=0; i<2; i++)
    {
      for (int j=0; j<3; j++)
      {
        mpsIndex subs[] = {i, j};
        mpsIndex id = mpsCalcSingleSubscript(outVal[0], 2, subs);
        std::cout << out[id] << "\t";
      }
      std::cout << std::endl;
    }
  }
  else
  {
    mpsErrorInfo error;
    mpsruntime->getLastErrorInfo(context, &error);
    std::cout << "Error: " << error.message << std::endl;

    switch(error.type)
    {
    case MPS_HTTP_ERROR_INFO:
      std::cout << "HTTP: "
          << error.details.http.responseCode
          << ": " << error.details.http.responseMessage
          << std::endl;
    case MPS_MATLAB_ERROR_INFO:
      std::cout << "MATLAB: " << error.details.matlab.identifier
          << std::endl;
      std::cout << error.details.matlab.message << std::endl;
    case MPS_GENERIC_ERROR_INFO:
      std::cout << "Generic: "
          << error.details.general.genericErrorMsg
          << std::endl;
    }
    mpsruntime->destroyLastErrorInfo(&error);
  }

  for (int i=0; i<numIn; i++)
    mpsDestroyArray(inVal[i]);
  delete[] inVal;

  for (int i=0; i<numOut; i++)
    mpsDestroyArray(outVal[i]);
  delete[] outVal;

  mpsruntime->destroyConfig(config);
  mpsruntime->destroyContext(context);
  mpsTerminate();
}
```

18 Compile the application.

To compile your client code, the compiler needs access to `client.h`. This header file is stored in *`matlabroot`*`/toolbox/compiler_sdk/mps_clients/c/include/mps/`.

To link your application, the linker needs access to the following files stored in *`matlabroot`*`/toolbox/compiler_sdk/mps_clients/c/`:

Files Required for Linking

Windows	UNIX®/Linux	Mac OS X
`$arch\lib\mpsclient.lib`	`$arch/lib/libprotobuf.so`	`$arch/lib/libprotobuf.dylib`
	`$arch/lib/libcurl.so`	`$arch/lib/libcurl.dylib`
	`$arch/lib/libmwmpsclient.so`	`$arch/lib/libmwmpsclient.dylib`
	`$arch/lib/libmwcpp11compat.so`	

19 Run the application.

To run your application, add the following files stored in *matlabroot*/toolbox/compiler_sdk/mps_clients/c/ to the application's path:

Files Required for Running

Windows	UNIX/Linux	Mac OS X
$arch\lib\mpsclient.dll	$arch/lib/libprotobuf.so	$arch/lib/libprotobuf.dylib
$arch\lib\libprotobuf.dll	$arch/lib/libcurl.so	$arch/lib/libcurl.dylib
$arch\lib\libcurl.dll	$arch/lib/libmwmpsclient.so	$arch/lib/libmwmpsclient.dylib
	$arch/lib/libmwcpp11compat.so	

The client invokes `addmatrix` function on the server instance and returns the following matrix at the console:

```
5.0 7.0 9.0
9.0 7.0 5.0
```

10

RESTful API JSON Encode and Decode Functions

mps.json.encode

Convert MATLAB data to JSON text using MATLAB Production Server JSON schema

Syntax

```
text = mps.json.encode(data)
text = mps.json.encode(data,Name,Value)
```

Description

`text = mps.json.encode(data)` encodes MATLAB data and returns JSON text in JSON schema for MATLAB Production Server. You can use this JSON text on multiple platforms to encode content for MATLAB Production Server.

`text = mps.json.encode(data,Name,Value)` specifies additional options with one or more name-value pair arguments for specific input cases. For example, you can decide to encode `data` in the large or small format defined for representing data types.

Examples

Convert a Matrix to JSON Schema for MATLAB Production Server

Encode a 3-by-3 magic square in the JSON format.

```
mps.json.encode(magic(3))
```

```
ans =
    '[[8,1,6],[3,5,7],[4,9,2]]'
```

Convert a Matrix and Specify Format for JSON Schema for MATLAB Production Server

Encode a 3-by-3 magic square in JSON using the `large` format option.

```
mps.json.encode(magic(3),'Format','large')
```

```
ans =
    '{"mwdata":[8,3,4,1,5,9,6,7,2],"mwsize":[3,3],"mwtype":"double"}'
```

Convert an Array Containing `NaN`, `Inf`, or `-Inf` to JSON Schema for MATLAB Production Server

Encode an array containing `-Inf`, `NaN`, and `Inf` in JSON using '`object`' in '`NanInfType`' option.

```
mps.json.encode([-Inf NaN Inf],'NaNInfType','object','Format','large')
```

```
ans =
    '{"mwdata":[{"mwdata":"-Inf"},{"mwdata":"NaN"},{"mwdata":"Inf"}], "mwsize":[1,3],"mwtype":"double"}'
```

Input Arguments

`data` — MATLAB data that MATLAB Production Server supports

numeric | character | logical | structure | cell

MATLAB data that MATLAB Production Server supports, specified as a numeric, character, logical, structure, or 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: `mps.json.encode(data,'Format','large')`

`Format` — Format to encode `data`

'small' (default) | 'large'

Format to encode MATLAB data, specified as the comma-separated pair consisting of '`Format`' and the format '`small`' or '`large`'.

The `small` format is a simpler representation of MATLAB data types in JSON, whereas the `large` format is a more generic representation. For more information, see “JSON Representation of MATLAB Data Types”.

`NaNInfType` — Format to encode `NaN`, `Inf`, and `-Inf` in `data`

'string' (default) | 'object'

Format to encode `NaN`, `Inf`, and `-Inf` in `data`, specified as a comma-separated pair consisting of '`NaNInfType`' and the JSON data-types '`string`' or '`object`'.

`PrettyPrint` — Format `text` for readability

false (default) | true

Format `text` for readability, specified as a comma-separated pair consisting of '`PrettyPrint`' and logical '`true`' or '`false`'.

`PrettyPrint` enables better readability for a user when set to `true`. Syntax is `mps.json.encode(magic(3),'PrettyPrint',true)`.

Output Arguments

`text` — JSON-formatted text

character vector

JSON-formatted text for JSON schema for MATLAB Production Server, returned as a character vector.

Version History

Introduced in R2018a

See Also

`mps.json.decode` | `mps.json.encoderequest` | `mps.json.decoderesponse`

Topics

“JSON Representation of MATLAB Data Types” (MATLAB Production Server)
“Create Deployable Archive for MATLAB Production Server” (MATLAB Production Server)

mps.json.decode

Convert a character vector or string in MATLAB Production Server JSON schema to MATLAB data

Syntax

```
data = mps.json.decode(text)
```

Description

`data = mps.json.decode(text)` parses JSON schema for MATLAB Production Server to convert it to MATLAB data.

Examples

Decode JSON-Formatted Text for a Matrix

```
mps.json.decode('[[8,1,6],[3,5,7],[4,9,2]]')
```

```
ans =
     8     1     6
     3     5     7
     4     9     2
```

Decode a Matrix in JSON That Uses `large` Format

```
mps.json.decode('{"mwdata":[1,4,3,2],"mwsize":[2,2],"mwtype":"double"}')
```

```
ans =
      1     3
      4     2
```

Input Arguments

`text` — JSON text following the schema for MATLAB Production Server

character vector (default) | string

JSON-formatted text that follows the schema for MATLAB Production Server, specified as a character vector or string.

`text` can be in various formats like `small`, `large`, `NaNInfType`, and `PrettyPrint`, as explained in “Name-Value Pair Arguments” on page 10-3 on the `mps.json.encode` page.

Output Arguments

`data` — MATLAB data

any MATLAB data type

MATLAB data decoded from MATLAB Production Server JSON text.

Version History

Introduced in R2018a

See Also

`mps.json.encode` | `mps.json.encoderequest` | `mps.json.decoderesponse`

Topics

“JSON Representation of MATLAB Data Types” (MATLAB Production Server)
“Create Deployable Archive for MATLAB Production Server” (MATLAB Production Server)

mps.json.encoderequest

Convert MATLAB data in a server request to JSON text using MATLAB Production Server JSON schema

Syntax

```
text = mps.json.encoderequest(rhs)
text = mps.json.encoderequest(rhs,Name,Value)
```

Description

`text = mps.json.encoderequest(rhs)` encodes the request that is input to the deployed MATLAB function using JSON schema for MATLAB Production Server. It builds a server request that includes MATLAB variables and options, such as `'Nargout'` and `'OutputFormat'`, that are needed to make a call to MATLAB Production Server.

`text = mps.json.encoderequest(rhs,Name,Value)` specifies additional options with one or more name-value pair arguments for specific input cases.

Examples

Write MATLAB Production Server Payload

```
mps.json.encoderequest({[1 2 3 4]})
```

```
ans =
    '{"rhs":[[[1,2,3,4]]],"nargout":1,"outputFormat":{"mode":"small","nanType":"string"}}'
```

Write MATLAB Production Server Payload, and Set Output Parameters

```
rhs = {['Red'], [15], [1 3; 5 7], ['Green']};
mps.json.encoderequest(rhs, 'Nargout', 3, 'OutputFormat', 'large')
```

```
ans =
    '{"rhs":["Red",15,[[1,3],[5,7]],"Green"],"nargout":3,"outputFormat":{"mode":"large","nanType":"string"}}'
```

Write a MATLAB Function as MATLAB Production Server Payload

Use the MATLAB function `horzcat` that horizontally concatenates two matrices.

```
a = [1 2; 5 6];
b = [3 4; 7 8];
mps.json.encoderequest({horzcat(a,b)})
```

```
ans =
    '{"rhs":[[[1,2,3,4],[5,6,7,8]]],"nargout":1,"outputFormat":{"mode":"small","nanType":"string"}}'
```

Read Response from a `sortstudent` Function Deployed on MATLAB Production Server

Execute `mps.json.encoderequest` and `mps.json.decoderesponse` to call a function deployed on MATLAB Production Server using `webwrite`. In this case, student names and their corresponding scores are deployed to MATLAB Production Server to the `sortstudents` function that sorts students based on their scores. The result returned is the equivalent to calling the function `sortstudents(struct('name', 'Ed', 'score', 83), struct('name', 'Toni', 'score', 91))` from MATLAB.

Assume that there is a deployable archive `studentapp` that contains a MATLAB function `sortstudents` deployed to the server.

```
data = {struct('name', 'Ed', 'score', 83), struct('name', 'Toni', 'score', 91)};
body = mps.json.encoderequest(data);

options = weboptions;

% Create a weboptions object that instructs webread to return JSON text
options.ContentType = 'text';

% Create a weboptions object that instructs webwrite to encode character vector data as JSON to post it to a web service
options.MediaType = 'application/json';

response = webwrite('http://localhost:9910/studentapp/sortstudents', body, options);

result = mps.json.decoderesponse(response);
```

Input Arguments

`rhs` — Input arguments for deployed MATLAB function that is called

cell vector of any MATLAB data type supported by MATLAB Production Server

Input arguments for a MATLAB function deployed on MATLAB Production Server that is called, specified as a cell vector.

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: `mps.json.encoderequest(rhs, 'Format', 'large')`

`Nargout` — Number of output arguments for function deployed on MATLAB Production Server

1 (default) | any positive integer

Number of output arguments for function deployed on MATLAB Production Server, specified as comma-separated pair consisting of '`Nargout`' and number of output arguments.

`mps.json.encoderequest(rhs, 'Nargout', 3)`.

`Format` — Format to encode `rhs`

'small' (default) | 'large'

Format to encode `rhs`, specified as comma-separated pair consisting of '`Format`' and the format '`small`' or '`large`'.

The `small` format is a simpler representation of MATLAB data types in JSON, whereas the `large` format is a more generic representation. For more information, see “JSON Representation of MATLAB Data Types”.

`NaNInfType` — Format to encode `NaN`, `Inf`, `-Inf` in `rhs`
'string' (default) | 'object'

Format to encode `NaN`, `Inf`, `-Inf` in `rhs`, specified as comma-separated pair consisting of '`NaNInfType`' and JSON data types '`string`' and '`object`'.

`OutputFormat` — Format for response from MATLAB function deployed on MATLAB Production Server
'small' (default) | 'large'

Format for response from MATLAB function deployed on MATLAB Production Server, specified as comma-separated pair consisting of '`OutputFormat`' and the format '`small`' or '`large`'.

Output format is set using `mps.json.encoderequest(rhs, 'OutputFormat', 'large')`.

`OutputNanInfType` — Type for response from MATLAB function deployed on MATLAB Production Server containing `NaN`, `Inf`, `-Inf`
'string' (default) | 'object'

Type for response from MATLAB function deployed on MATLAB Production Server containing `NaN`, `Inf`, `-Inf`, specified as comma-separated pair consisting of '`OutputNaNInfType`' and JSON data type '`string`' and '`object`'.

NaN-type for output response is set using `mps.json.encoderequest(rhs, 'OutputNaNInfType', 'object')`.

`PrettyPrint` — Format `text` for readability
false (default) | true

Format `text` for readability, specified as a comma-separated pair consisting of '`PrettyPrint`' and logical '`true`' or '`false`'. Syntax is `mps.json.encoderequest(rhs,'PrettyPrint',true)`.

Output Arguments

`text` — JSON text
character vector

JSON-formatted text for JSON schema for MATLAB Production Server, returned as a character vector.

Version History

Introduced in R2018a

See Also

`mps.json.encode` | `mps.json.decode` | `mps.json.decoderesponse`

Topics
“JSON Representation of MATLAB Data Types” (MATLAB Production Server)

“Create Deployable Archive for MATLAB Production Server” (MATLAB Production Server)

mps.json.decoderesponse

Convert JSON text from a server response to MATLAB data

Syntax

```
lhs = mps.json.decoderesponse(response)
error = mps.json.decoderesponse(response)
```

Description

`lhs = mps.json.decoderesponse(response)` reads the JSON payload of the output arguments returned from a successful MATLAB function call.

`error = mps.json.decoderesponse(response)` reads the JSON payload of the MATLAB error thrown from a failed MATLAB function call.

Examples

Read from MATLAB Production Server Payload

```
mps.json.decoderesponse('{"lhs":[[[1, 2, 3, 4]]]}')
```

```
ans =
  1x1 cell array
    {1x4 double}
```

Read response from a `sortstudent` function deployed on MATLAB Production Server

Execute `mps.json.encoderequest` and `mps.json.decoderesponse` to call a function deployed on MATLAB Production Server using `webwrite`. In this case, student names and their corresponding scores are deployed to MATLAB Production Server to the `sortstudents` function that sorts students based on their scores. The result returned is the equivalent to calling the function `sortstudents(struct('name', 'Ed', 'score', 83), struct('name', 'Toni', 'score', 91))` from MATLAB.

Assume that there is a deployable archive `studentapp` that contains a MATLAB function `sortstudents` deployed to the server.

```
data = {struct('name', 'Ed', 'score', 83), struct('name', 'Toni', 'score', 91)};
body = mps.json.encoderequest(data);

options = weboptions;

% Create a weboptions object that instructs webread to return JSON text
options.ContentType = 'text';

% Create a weboptions object that instructs webwrite to encode character vector data as JSON to post it to a web service
options.MediaType = 'application/json';

response = webwrite('http://localhost:9910/studentapp/sortstudents', body, options);
```

```
result = mps.json.decoderesponse(response);
```

Input Arguments

`response` — JSON result from a MATLAB function call

char (default)

JSON result from a MATLAB function call specified as JSON text.

Output Arguments

`lhs` — Cell vector of output arguments

`Cell` vector

Cell vector of output arguments that are from a MATLAB function called from MATLAB Production Server.

`error` — Generated output when request results in a MATLAB error

`struct` array

Generated output when request to MATLAB function called from MATLAB Production Server results in a MATLAB error returned as a `struct` array.

Version History

Introduced in R2018a

See Also

`mps.json.encode` | `mps.json.decode` | `mps.json.encoderequest`

Topics

“JSON Representation of MATLAB Data Types” (MATLAB Production Server)
“Create Deployable Archive for MATLAB Production Server” (MATLAB Production Server)

prodserver.metrics.incrementCounter

Create Prometheus `counter` metric

Syntax

```
prodserver.metrics.incrementCounter(metricName,metricValue)
```

Description

`prodserver.metrics.incrementCounter(metricName,metricValue)` creates a custom Prometheus® counter metric. Prometheus counter values can only increase over time. The metric is created when the following conditions are true:

- `prodserver.metrics.incrementCounter` is present in the MATLAB function that you deploy to MATLAB Production Server.
- A client invokes the deployed MATLAB function that contains `prodserver.metrics.incrementCounter`.

The server collects the metric when the deployed MATLAB function executes. The output of the GET Metrics (MATLAB Production Server) API returns information about the metric name and the metric value.

Examples

Create Custom Prometheus Counter Metric

Create a custom counter metric that a Prometheus server can monitor.

Write a MATLAB function that increments the counter. In practice, you create metrics related to your application that help you instrument your code.

```
function rc = test_metric_value()
prodserver.metrics.incrementCounter("test_requests_processed",1);
rc = 0;
end
```

Package and deploy the MATLAB function to the server.

When a client executes the deployed function, the value of the `test_requests_processed` metric is incremented by 1.

For a detailed example, see “Create Custom Prometheus Metrics”.

Input Arguments

`metricName` — Name of Prometheus counter metric
character array | string scalar

Name of the Prometheus counter metric, specified as a character array or string scalar. The name must be a valid MATLAB variable name.

Example: test_requests_processed

`metricValue` — Value of counter
positive numeric scalar | `Inf`

Numeric value of the counter metric, specified as a scalar. The value must be positive. The value can only increase over time.

Example: 1

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

Version History

Introduced in R2022a

See Also

`prodserver.metrics.setGauge`

Topics
“Metrics Service” (MATLAB Production Server)
GET Metrics (MATLAB Production Server)

External Websites
Prometheus Metric Types

prodserver.metrics.setGauge

Create Prometheus `gauge` metric

Syntax

```
prodserver.metrics.setGauge(metricName,metricValue)
```

Description

`prodserver.metrics.setGauge(metricName,metricValue)` creates a custom Prometheus gauge metric. Prometheus gauge values can increase or decrease over time. The metric is created when the following conditions are true:

- `prodserver.metrics.setGauge` is present in the MATLAB function that you deploy to MATLAB Production Server.
- A client invokes the deployed MATLAB function that contains `prodserver.metrics.setGauge`.

The server collects the metric when the deployed MATLAB function executes. The output of the GET Metrics (MATLAB Production Server) API returns information about the metric name and the metric value.

Examples

Create Custom Prometheus Gauge Metric

Create a custom gauge metric that a Prometheus server can monitor.

Write a MATLAB function that sets the gauge to a specific value. In practice, you create metrics related to your application that help you instrument your code.

```
function rc = test_metric_value()
prodserver.metrics.setGauge("requests_in_progress",4);
rc = 0;
end
```

Package and deploy the MATLAB function to the server.

When a client executes the deployed function, the value of the `requests_in_progress` metric is set to 4.

For a detailed example, see “Create Custom Prometheus Metrics”.

Input Arguments

`metricName` — Name of Prometheus gauge metric

character array | string scalar

Name of the Prometheus gauge metric, specified as a character array or string scalar. The name must be a valid MATLAB variable name.

Example: requests_in_progress

`metricValue` — Value of gauge
numeric scalar | `-Inf` | `Inf` | `NaN`

Numeric value of the gauge metric, specified as a scalar. The value can increase or decrease over time.

Example: 4

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

Version History

Introduced in R2022a

See Also

`prodserver.metrics.incrementCounter`

Topics
“Metrics Service” (MATLAB Production Server)
GET Metrics (MATLAB Production Server)

External Websites
Prometheus Metric Types

Persistence Functions

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 `start` 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” (MATLAB Production Server)

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” (MATLAB Production Server)

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` | `release`

Topics

“Data Caching Basics” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

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” (MATLAB Production Server)

Examples

Deploy Object Detection Model as Microservice

This example shows how to create a microservice Docker image from a MATLAB object detection model. The microservice image created by MATLAB Compiler SDK provides an HTTP/HTTPS endpoint to access MATLAB code.

You package a MATLAB function into a deployable archive, and then create a Docker image that contains the archive and a minimal MATLAB Runtime package. You can then run the image in Docker and make calls to the service using any of the MATLAB Production Server client APIs.

Download Support Package

Type `matlabshared.supportpkg.getInstalled` at the MATLAB command prompt to verify whether the following add-on is installed:

- Computer Vision Toolbox Model for YOLO v4 Object Detection

If you need to install the add-on, click the **Add-Ons** icon in the MATLAB toolstrip and search for the add-on. You can also download and install it from the MathWorks File Exchange.

Prerequisites

- Verify that you have MATLAB Compiler SDK installed on the development machine.
- Verify that you have Docker installed and configured on the development machine by typing `[~,msg] = system('docker version')` in a MATLAB command window. If you are using WSL, use the command `[~,msg] = system('wsl docker version')` instead.
- If you do not have Docker installed, follow the instructions on the Docker website to install and set up Docker. For details, see `docs.docker.com/engine/install/`.
- To build microservice images on Windows, you must install either Docker Desktop or Docker on Windows Subsystem for Linux v2 (WSL2). To install Docker Desktop, see `docs.docker.com/desktop/windows/install/`. For instructions on how to install Docker on WSL2, see `https://www.mathworks.com/matlabcentral/answers/1758410-how-do-i-install-docker-on-wsl2`.
- If the computer you are using is not connected to the Internet, you must download the MATLAB Runtime installer for Linux from a computer that is connected to the Internet and transfer the installer to the computer that is not connected to the Internet. Then, on the offline machine, run the command `compiler.runtime.createInstallerDockerImage`, where `filepath` is the path to the MATLAB Runtime installer archive. You can download the installer from the MathWorks website. For details, see `https://www.mathworks.com/products/compiler/matlab-runtime.html`.

Create MATLAB Function to Detect Objects

Write an object detection function named `cvt` using the following code. Save the function in a file named `cvt.m`.

```
function [bboxes, scores, labels] = cvt(imageUrl)
iminfo = imfinfo(imageUrl);
    % Read image
    % If indexed image, read colormap and convert to rgb
    if strcmp(iminfo.ColorType,'indexed') == 1
        [im, cmap] = webread(imageUrl, 'Timeout', 10);s(
```

```
        im = ind2rgb(im, cmap);
    else
        im = webread(imageUrl, 'Timeout', 10);
    end
% Add  pretrained YOLO v4 dataset tinyYOLOv4COCO.mat to MATLAB path for testing
% Comment or remove the next 2 lines of code prior to deploying as microservice
detectorPath = [matlabshared.supportpkg.getSupportPackageRoot, '/toolbox/vision/supportpackages/y
addpath(detectorPath)
load('tinyYOLOv4COCO.mat', 'detector');

% Detect objects in image using detector
[bboxes,scores,labels] = detect(detector,im);
labels = cellstr(labels);
end
```

Test the function from the MATLAB command line:

```
%% Specify image URL
imageUrl = "https://www.mathworks.com/help/examples/deeplearning_shared/win64/TrafficSignDetectio
%% Display image
imageFile = "trafficimage.jpg";
imageFileFullPath = websave(imageFile, imageUrl);
[im, cmap] = imread(imageFileFullPath);
imshow(im, cmap)
%% Detect objects in image
[bboxes, scores, labels] = cvt(imageUrl)
```

```
bboxes =
  2×4 single matrix
  445.3871  326.4009  223.3270   98.7086
  504.2861  271.4571   45.7471   41.0955
scores =
  2×1 single column vector
    0.9151
    0.6610
labels =
  2×1 cell array
    {'truck'    }
    {'stop sign'}
```

Create Deployable Archive

Comment the following lines of code in the `cvt.m` file prior to creating a deployable archive.

```
% detectorPath = [matlabshared.supportpkg.getSupportPackageRoot, '/toolbox/vision/supportpackages
% addpath(detectorPath)
```

Package the `cvt` function into a deployable archive using the `compiler.build.productionServerArchive` function.

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

```
buildResults = compiler.build.productionServerArchive('cvt.m', ...
    'ArchiveName','yolov4od','Verbose',true, ...
    'SupportPackages',{'Computer Vision Toolbox Model for YOLO v4 Object Detection'};
```

```
buildResults =
  Results with properties:
```

```
                BuildType: 'productionServerArchive'
                    Files: {'/home/mluser/work/yolov4odproductionServerArchive/yolov4od.ctf'}
IncludedSupportPackages: {'Computer Vision Toolbox Model for YOLO v4 Object Detection'}
                  Options: [1×1 compiler.build.ProductionServerArchiveOptions]
```

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

Once the build is complete, the function creates a folder named `yolov4odproductionServerArchive` in your current directory to store the deployable archive.

Package Archive into Microservice Docker Image

Build the microservice Docker image using the `buildResults` object that you created.

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

```
compiler.package.microserviceDockerImage(buildResults,...
    'ImageName','yolov4od-microservice',...
    'DockerContext',fullfile(pwd,'microserviceDockerContext'));
```

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

- `applicationFilesForMATLABCompiler/yolov4od.ctf` — Deployable archive file.
- `Dockerfile` — Docker file that specifies Docker run-time options.
- `GettingStarted.txt` — Text file that contains deployment information.

Test Docker Image

In a system command window, verify that your `yolov4od-microservice` image is in your list of Docker images.

```
docker images
```

```
REPOSITORY                                        TAG           IMAGE ID            CREATED
yolov4od-microservice                             latest        4401fa2bc057        33 seconds ago
matlabruntime/r2022b/update0/4200000000000000     latest        5259656e4a32        24 minutes ago
```

Run the `yolov4od-microservice` microservice image from the system command prompt.

```
docker run --rm -p 9900:9910 yolov4od-microservice -l trace &
```

Port 9910 is the default port exposed by the microservice within the Docker container. You can map it to any available port on your host machine. For this example, it is mapped to port 9900.

You can specify additional options in the Docker command. For a complete list of options, see “Microservice Command Arguments” on page 1-17.

Once the microservice container is running in Docker, you can check the status of the service by going to the following URL in a web browser: `http://`*`hostname`*`:9900/api/health`

If the service is ready to receive requests, you see the following message: `"status: ok"`

Test the running service. In the terminal, use the `curl` command to send a JSON query with the input argument 4 to the service through port 9900. For more information on constructing JSON requests,

see “JSON Representation of MATLAB Data Types” (MATLAB Production Server) (MATLAB Production Server).

```
curl -v -H Content-Type:application/json \
-d '{"nargout":3,"rhs":["https://www.mathworks.com/help/examples/deeplearning_shared/win64/Traffi
"http://hostname:9900/yolov4od/cvt" | jq -c
```

The output is:

```
{"lhs":[{"mwdata":[445.387146,504.286102,326.40094,271.457092,223.327026,45.7471,98.7086487,41.09
{"mwdata":[0.91510725,0.661022],"mwsize":[2,1],"mwtype":"single"},
{"mwdata":[{"mwdata":["truck"],"mwsize":[1,5],"mwtype":"char"},
{"mwdata":["stop sign"],"mwsize":[1,9],"mwtype":"char"}],"mwsize":[2,1],"mwtype":"cell"}]}
```

You can also test from the MATLAB desktop:

```
%% Import MATLAB HTTP interface packages
import matlab.net.*
import matlab.net.http.*
import matlab.net.http.fields.*

%% Setup message body
body = MessageBody;
body.Payload = ...
    '{"nargout": 3,"rhs": ["https://www.mathworks.com/help/examples/deeplearning_shared/win64/Tra

%% Setup request
requestUri = URI('http://hostname:9900/yolov4od/cvt');
options = matlab.net.http.HTTPOptions('ConnectTimeout',20,...
    'ConvertResponse',false);
request = RequestMessage;
request.Header = HeaderField('Content-Type','application/json');
request.Method = 'POST';
request.Body = body;

%% Send request & view raw response
response = request.send(requestUri, options);
disp(response.Body.Data)

%% Decode JSON
lhs = mps.json.decoderesponse(response.Body.Data);

%% Clean up printed output
for i = 1:length(lhs)
    [r,c] = size(lhs{i});
    if ~iscell(lhs{i}) && c==1
        tmp(:,i) = num2cell(lhs{i});
    elseif ~iscell(lhs{i}) && c~=1
        tmp(:,i) = num2cell(lhs{i},2);
    else
        tmp(:,i) = lhs{i};
    end
end
%% Display response as a table
T = cell2table(tmp,'VariableNames',{'Boxes', 'Scores', 'Labels'})
```

The output is:

```
T =

  2×3 table

                  Boxes                      Scores        Labels
    ____________________________________    _______    _____________

    445.39     326.4    223.33    98.709    0.91511    {'truck'    }
    504.29    271.46    45.747    41.096    0.66102    {'stop sign'}
```

To stop the service, use the following command to display the container id.

```
docker ps
```

```
CONTAINER ID        IMAGE                    COMMAND                  CREATED             STATUS
f372b8b574e8        yolov4od-microservice    "/opt/matlabruntime/…"   6 hours ago         Up 6 hou
```

Stop the service using the specified container id.

```
docker stop f372b8b574e8
```

Share Docker Image

You can share your Docker image in various ways.

- Push your image to the Docker central registry DockerHub, or to your private registry. This is the most common workflow.
- Save your image as a tar archive and share it with others. This workflow is suitable for immediate testing.

For details about pushing your image to DockerHub or your private registry, consult the Docker documentation.

Save Docker Image as Tar Archive

To save your Docker image as a tar archive, open a system command window, navigate to the Docker context folder, and type the following.

```
docker save yolov4od-microservice -o yolov4od-microservice.tar
```

This command creates a file named `yolov4od-microservice.tar` in the current folder. Set the appropriate permissions (for example, using `chmod`) prior to sharing the tarball with other users.

Load Docker Image from Tar Archive

Load the image contained in the tarball on the end user machine.

```
docker load --input yolov4od-microservice.tar
```

Verify that the image is loaded.

```
docker images
```

Run Docker Image

```
docker run --rm -p 9900:9910 yolov4od-microservice
```

See Also

`matlabshared.supportpkg.getInstalled` | `compiler.build.productionServerArchive` | `compiler.package.microserviceDockerImage` | `compiler.runtime.createInstallerDockerImage`

External Websites

- https://docs.docker.com/engine/install/
- https://docs.docker.com/desktop/windows/install/
- https://www.mathworks.com/matlabcentral/answers/1758410-how-do-i-install-docker-on-wsl2
- https://www.mathworks.com/products/compiler/matlab-runtime.html