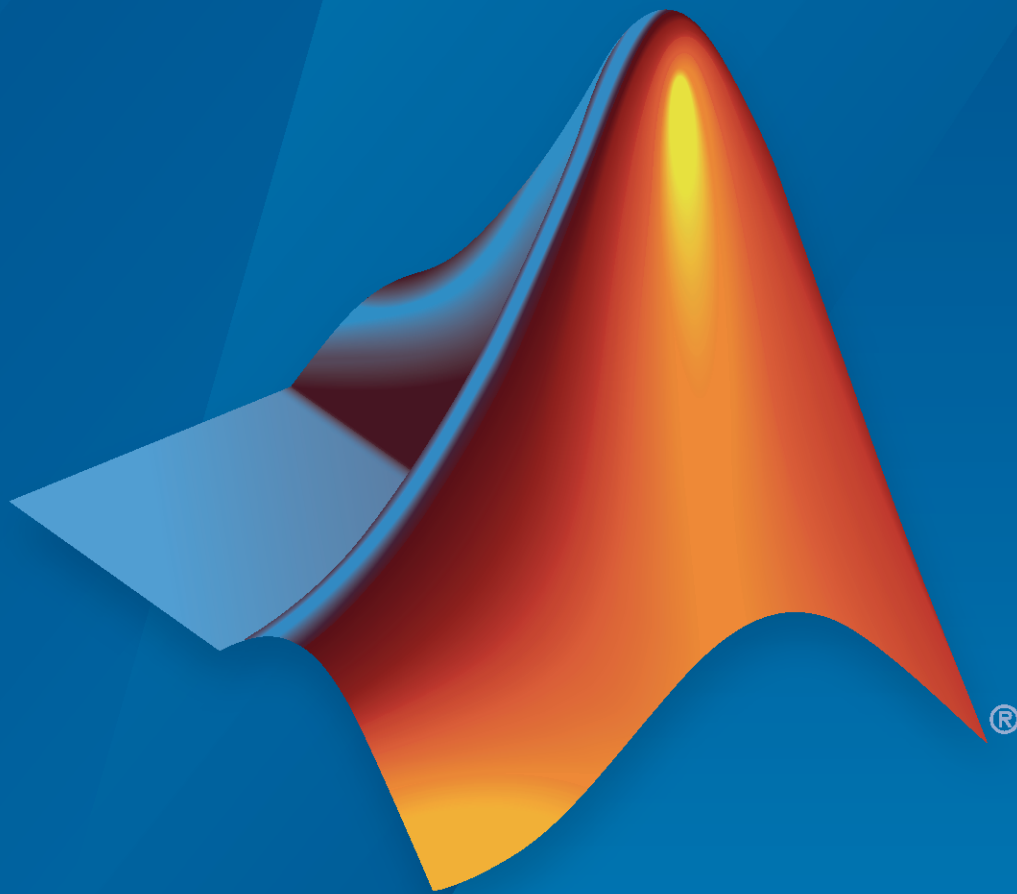


**MATLAB<sup>®</sup> Compiler SDK<sup>™</sup>**

MATLAB<sup>®</sup> Production Server<sup>™</sup> Testing Guide



**MATLAB<sup>®</sup>**

R2021a



## How to Contact MathWorks



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



Phone: 508-647-7000



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

*MATLAB® Compiler SDK™ MATLAB® Production Server™ Testing Guide*

© COPYRIGHT 2012–2021 by The MathWorks, Inc.

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

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

### Trademarks

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

### Patents

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

### Revision History

March 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)

	<b>Deployable Archive Creation</b>	
<b>1</b>		
	<b>Create Deployable Archive for MATLAB Production Server</b> .....	<b>1-2</b>
	Create MATLAB Function .....	<b>1-2</b>
	Create Deployable Archive with Production Server Compiler App .....	<b>1-2</b>
	Customize Application and Its Appearance .....	<b>1-3</b>
	Package Application .....	<b>1-3</b>
	<b>Create and Install a Deployable Archive with Excel Integration For MATLAB Production Server</b> .....	<b>1-5</b>
	Create Function In MATLAB .....	<b>1-5</b>
	Create Deployable Archive with Excel Integration Using Production Server Compiler App .....	<b>1-5</b>
	Customize the Application and Its Appearance .....	<b>1-6</b>
	Package the Application .....	<b>1-7</b>
	Install the Deployable Archive with Excel Integration .....	<b>1-8</b>
	<b>MATLAB Production Server Integration Testing</b>	
<b>2</b>		
	<b>Write a Test Client</b> .....	<b>2-2</b>
	<b>Test Client Data Integration Against MATLAB</b> .....	<b>2-3</b>
	Create a MATLAB Function .....	<b>2-3</b>
	Prepare for Testing .....	<b>2-3</b>
	Test Using RESTful API .....	<b>2-6</b>
	Testing Using Java Client Application .....	<b>2-10</b>
	<b>MATLAB Production Server Excel Add-In</b>	
<b>3</b>		
	<b>Data Marshaling Rules</b> .....	<b>3-2</b>
	Default Marshaling Rules .....	<b>3-2</b>
	Change Rules for Marshaling Data into MATLAB .....	<b>3-2</b>
	Change Rules for Marshaling Data into Excel .....	<b>3-2</b>

## MATLAB Production Server Excel Add-In

<b>4</b>	
	<b>XLA File Not Generated</b> ..... 4-2
	<b>Server Configuration Add-in Not Enabled</b> ..... 4-3
	<b>Error Using a Variable Number of Outputs</b> ..... 4-4

## Functions

**5**

## Apps

**6**

## Client Programming

**7**

	<b>Create a Java Client Using the MWHttpClient Class</b> ..... 7-2
	<b>Create a C# Client Using MWHttpClient</b> ..... 7-5
	<b>Create a Python Client</b> ..... 7-8
	<b>Create a C++ Client</b> ..... 7-9

## RESTful API JSON Encode and Decode Functions

**8**

## Persistence Functions

**9**

# Deployable Archive Creation

---

## Create Deployable Archive for MATLAB Production Server

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

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

### Create MATLAB Function

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

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

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

a = a1 + a2;
```

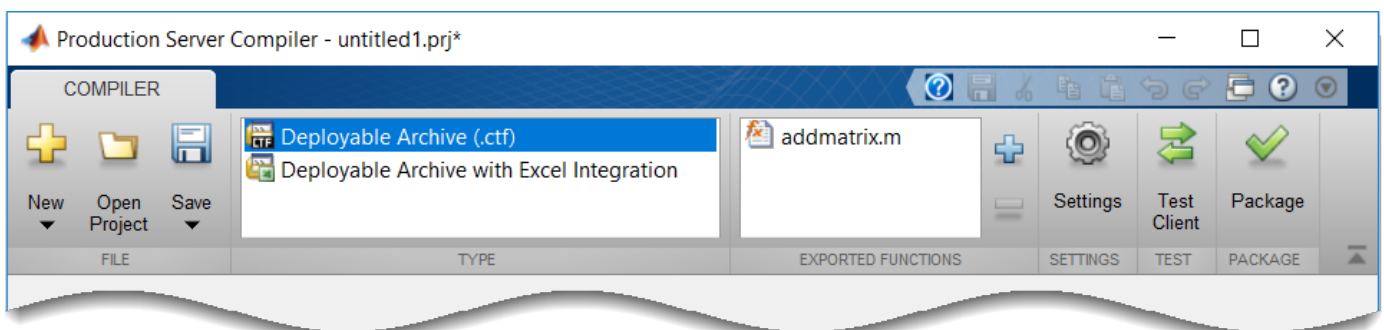
At the MATLAB command prompt, enter `addmatrix([1 4 7; 2 5 8; 3 6 9], [1 4 7; 2 5 8; 3 6 9])`.

The output is:


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

### Create Deployable Archive with Production Server Compiler App

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



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

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

Click **Open**.

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

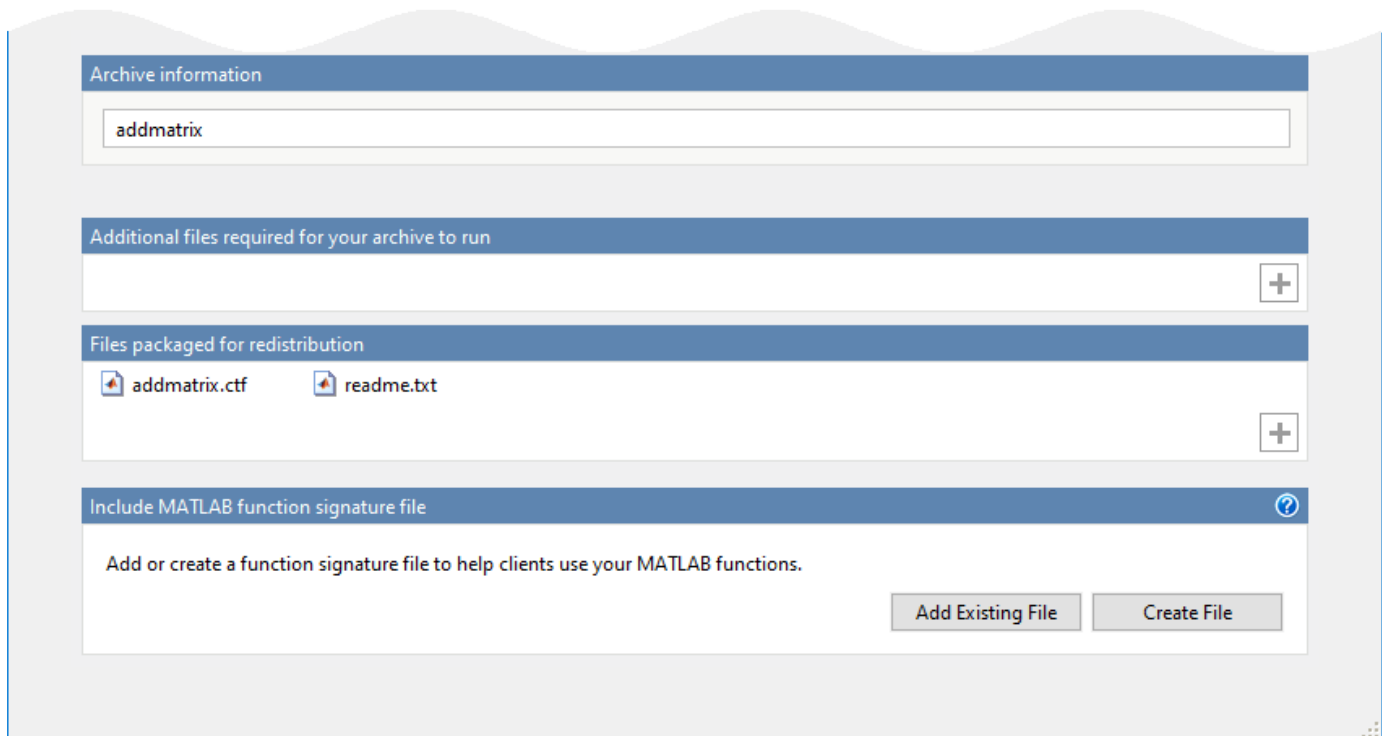
## Customize Application and Its Appearance

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

- **Archive information** — Editable information about the deployed archive.
- **Additional files required for your archive to run** — Additional files required to run the generated archive. These files are included in the generated archive installer. See “Manage Required Files in Compiler Project”.
- **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”.

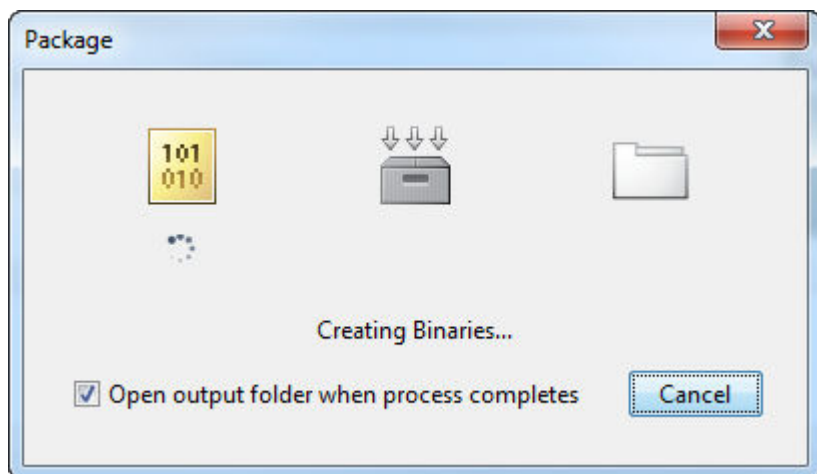
- **Include MATLAB function signature file** — Add or create a function signature file to help clients use your MATLAB functions. See “MATLAB Function Signatures in JSON”.



## 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.txt` — Log file generated by MATLAB Compiler™

## See Also

`deploytool` | `mcc` | `productionServerCompiler`

## More About

- Production Server Compiler
- “MATLAB Function Signatures in JSON”



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

**Supported platform:** Windows

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

## 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);
```

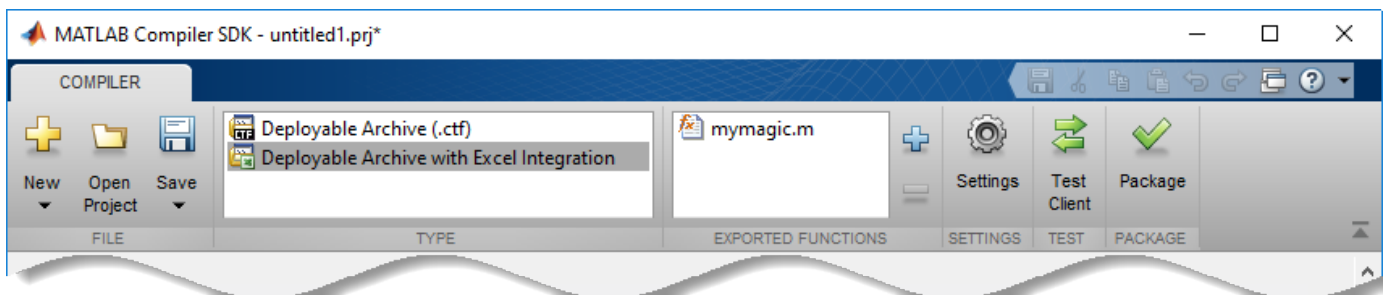
At the MATLAB command prompt, enter `mymagic(3)`.

The output is:

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

## Create Deployable Archive with Excel Integration Using Production Server Compiler App

- 1 On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Production Server Compiler**. In the **MATLAB Compiler SDK** project window, click **Deployable Archive with Excel integration**.



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

- 2 In the **MATLAB Compiler SDK** project window, specify the files of the MATLAB application that you want to deploy.

- 1 In the **Exported Functions** section, click .

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

Click **Open**.

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

## Customize the Application and Its Appearance

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

- **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)
  - 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.

**Archive information**

mymagic 1.0

Class Name	Method Name
Class1	[y] = mymagic (x) <span style="float: right;">+</span>

**Client configuration**

**Default Server URL**

None

MATLAB Production Server URL: Protocol:  Host:  Port:

Provide your own URL:

**Advanced Options**

Time the client waits before it times out:  Seconds

Maximum size of the response the client accepts:  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**

mymagic.ctf
 readme.txt
+

**Client**

mymagic.bas
 mymagic.dll
 mymagic.xla
 readme.txt
 ServerConfig.dll
+

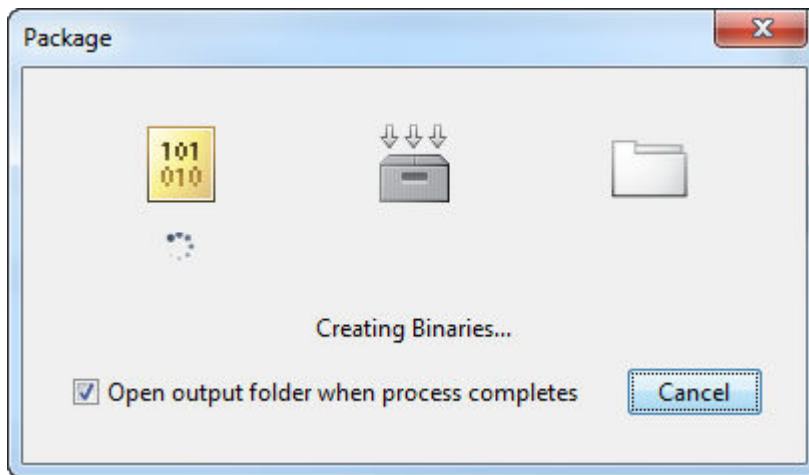
**Options**

Register the resulting component for you only 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.txt` — Log file generated by MATLAB Compiler

## Install the Deployable Archive with Excel Integration

The archive must be deployed to a MATLAB Production Server instance before the add-in works.

To install the deployable archive on a server instance:

- 1 Locate the archive in the `for_redistribution_files_only\server\` folder.  
The file name is similar to `archiveName.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

`productionServerCompiler`

# MATLAB Production Server Integration Testing

---

- “Write a Test Client” on page 2-2
- “Test Client Data Integration Against MATLAB” on page 2-3

## 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 a Java Client Using the MWHttpClient Class” on page 7-2
- “Create a C# Client Using MWHttpClient” on page 7-5
- “Create a Python Client” on page 7-8
- “Create a C++ Client” on page 7-9

## Test Client Data Integration Against MATLAB

### In this section...

“Create a MATLAB Function” on page 2-3  
 “Prepare for Testing” on page 2-3  
 “Test Using RESTful API” on page 2-6  
 “Testing Using Java Client Application” on page 2-10

This example shows you how to test your RESTful API or Java® client for deployment against MATLAB Production Server using the testing interface in the **Production Server Compiler** app. For testing purposes, you will create and use MATLAB function called `addmatrix` that accepts two numeric matrices as inputs and returns their sum as an output.

The testing interface can be accessed by clicking the **Test Client** button in the **Production Server Compiler** app. The **Production Server Compiler** app is part of MATLAB Compiler SDK.

### 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`.

#### **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)
```

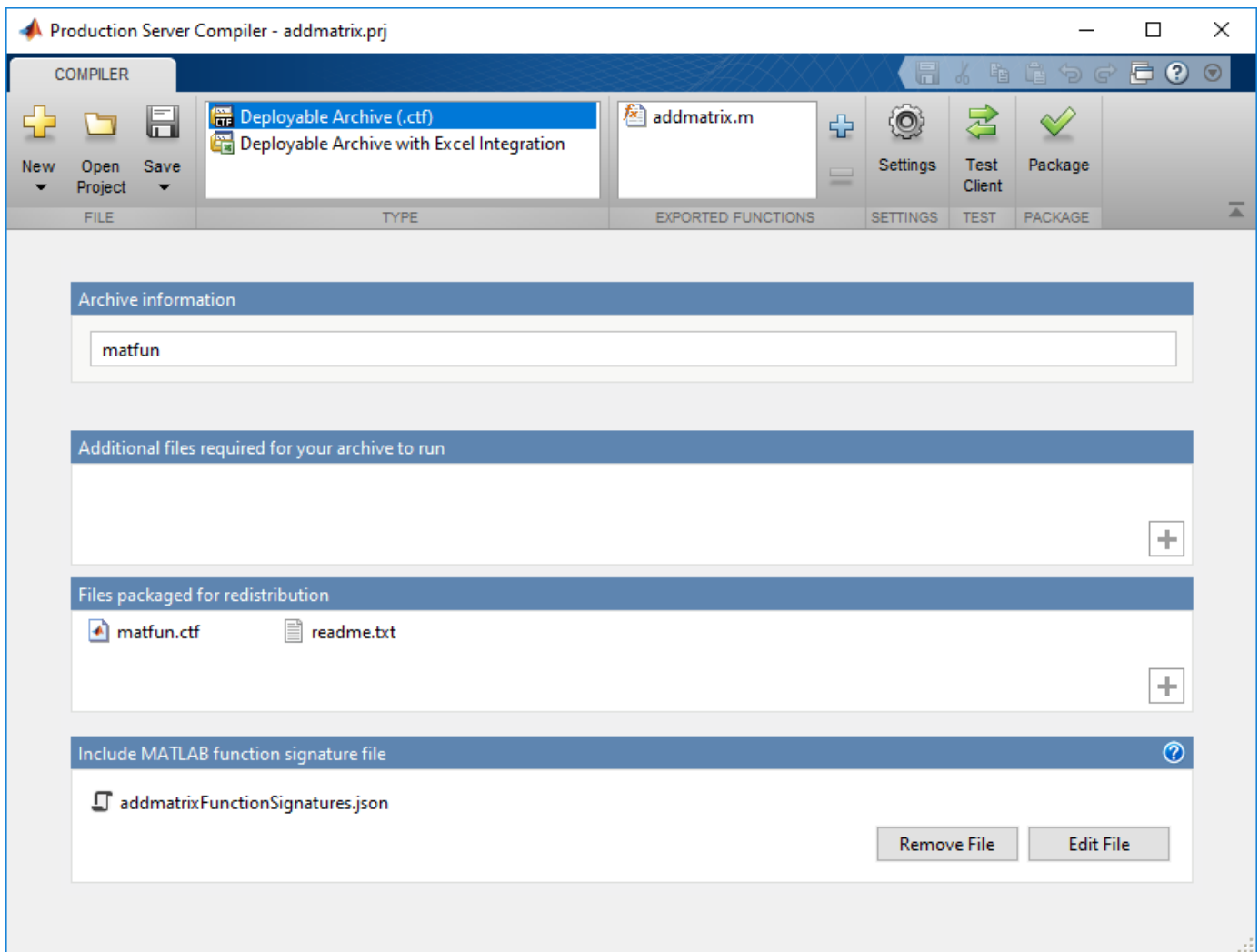
```
c =
```

```
    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 toolbar, select **Deployable Archive (.ctf)** from the list.
- 3 Specify the MATLAB functions to deploy.
  - a In the **Exported Functions** section of the toolbar, 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.

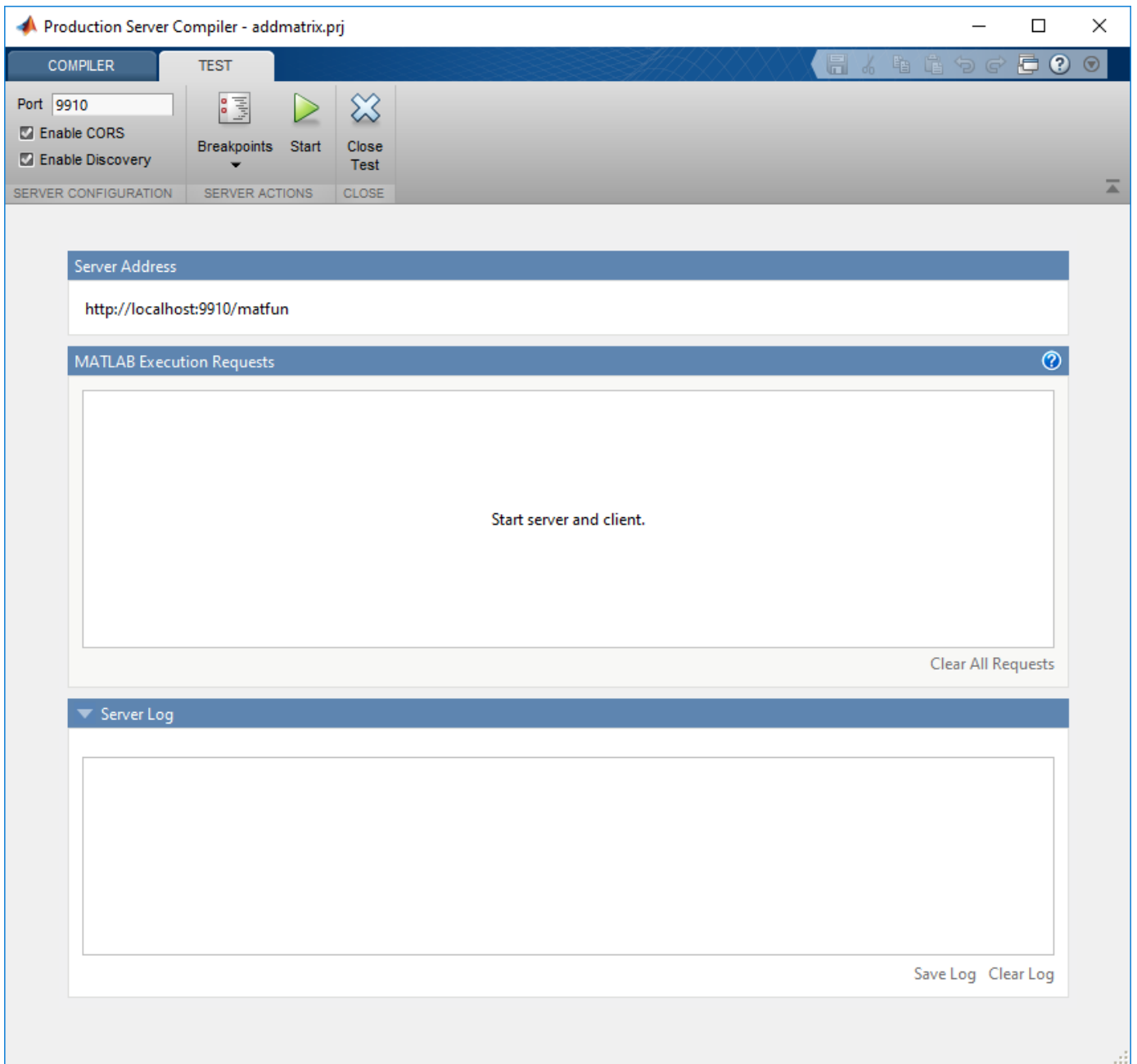
---

**Note** Only the MATLAB Production Server RESTful API supports the discovery service. For more information, see “RESTful API” (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 “HTTP Interface” 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';
```

- 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",
            "mwsizem": [],
            "mwtype": "double",
            "name": "a1"
          },
          {
            "help": "input matrix 2",
            "mwsizem": [],
            "mwtype": "double",
            "name": "a2"
          }
        ],
        "outputs": [
          {
            "help": "output matrix",
            "mwsizem": [],
            "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. This is because you cannot send a POST request from the same MATLAB session that is running the testing interface.
- 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.

ID	Function	Status
0	[a]=addmatrix(a1,a2)	✓ Complete

Input				Output			
Name	Size	Bytes	Class	Name	Size	Bytes	Class
a1	2x3	48	double array	a	2x3	48	double array
a2	2x3	48	double array				

Clear All Requests

- 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 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.

The screenshot shows the MATLAB IDE with the following components:

- Editor:** A function definition for `function a = addmatrix(a1, a2)` with a breakpoint (red circle) on the first line. The second line is `a = a1 + a2;`.
- Workspace:** A table showing variables `a1` and `a2` with their values.
 

Name	Value
a1	[10,20,30;40,50,60]
a2	[100,200,300;400,500,600]
- Command Window:** Shows the command `2 a = a1 + a2;` and the prompt `fx K>>`.

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 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 2-6.

## See Also

### Related Examples

- “Write a Test Client” on page 2-2
- “Package Deployable Archives with Production Server Compiler App”





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

---

## Data Marshaling Rules

In this section...
“Default Marshaling Rules” on page 3-2
“Change Rules for Marshaling Data into MATLAB” on page 3-2
“Change Rules for Marshaling Data into Excel” on page 3-2

### 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® #QNANs.
- 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.

## See Also



# **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.productionServerArchive

Create an archive for deployment to MATLAB Production Server

### Syntax

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

### Description

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

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

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

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

### Examples

#### Create Production Server Archive

Create a deployable server archive.

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

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

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

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

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

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

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

## Customize Production Server Archive

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

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

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

## Create Multiple Production Server Archives Using Options Object

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

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

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

```
opts =
```

ProductionServerArchiveOptions with properties:

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

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

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

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

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

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

### Get Build Information from Production Server Archive

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

Compile using the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

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

```
results =
```

Results with properties:

```
BuildType: 'productionServerArchive'
Files: 'D:\Documents\MATLAB\work\magicsquareproductionServerArchive\magicsquare.'
Options: [1x1 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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Verbose', 'on'`

### ArchiveName — Name of deployable archive

character vector | string scalar

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

Example: `'ArchiveName', 'MyMagic'`

Data Types: `char` | `string`

### AutoDetectDataFiles — Flag to automatically include data files

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

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

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

Example: `'AutoDetectDataFiles','off'`

Data Types: `logical`

### **FunctionSignatures — Path to JSON file**

character vector | string scalar

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

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

Data Types: `char` | `string`

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

character vector | string scalar

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

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

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

### **Verbose — Build verbosity**

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

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

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

Example: `'Verbose','off'`

Data Types: `logical`

## Output Arguments

### **results** — Build results

`compiler.build.Results` object

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

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

### **See Also**

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

**Introduced in R2020b**

# compiler.build.ProductionServerArchiveOptions

Options for building deployable archives

## Syntax

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

## Description

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

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

## Examples

### Create Deployable Archive Options Object

Create a `ProductionServerArchiveOptions` object from a function file.

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

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

```
opts =
```

`ProductionServerArchiveOptions` with properties:

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

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

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

```
opts =
```

ProductionServerArchiveOptions with properties:

```

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

```

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

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

### Customize Deployable Archive Options Object

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

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

```

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

```

```
opts =
```

ProductionServerArchiveOptions with properties:

```

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

```

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

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

```
opts =
```

ProductionServerArchiveOptions with properties:

```

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

```

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



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

## Input Arguments

### FunctionFiles — Files implementing MATLAB functions

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

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

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

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

### Name-Value Pair Arguments

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

Example: `'Verbose', 'on'`

### ArchiveName — Name of deployable archive

character vector | string scalar

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

Example: `'ArchiveName', 'MyMagic'`

Data Types: `char` | `string`

### AutoDetectDataFiles — Flag to automatically include data files

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

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

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

Example: `'AutoDetectDataFiles', 'off'`

Data Types: `logical`

### FunctionSignatures — Path to JSON file

character vector | string scalar

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

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

Data Types: char | string

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

character vector | string scalar

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

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

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

### **Verbose — Build verbosity**

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

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

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

Example: 'Verbose', 'off'

Data Types: logical

## **Output Arguments**

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

`ProductionServerArchiveOptions` object

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

## **See Also**

`productionServerCompiler`

**Introduced in R2020b**

# compiler.build.Results

Compiler build results object

## Description

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

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

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 5-13).
- Create a COM component using `compiler.build.comComponent` (example on page 5-14).
- Create a C shared library using `compiler.build.cSharedLibrary` (example on page 5-14).
- Create a C++ shared library using `compiler.build.cppSharedLibrary` (example on page 5-14).
- Create a .NET assembly using `compiler.build.dotNETAssembly` (example on page 5-15).
- Create a Java package using `compiler.build.javaPackage` (example on page 5-15).
- Create a Python® package using `compiler.build.pythonPackage` (example on page 5-16).

## Properties

### BuildType — Build type

'productionServerArchive' | 'comComponent' | 'cSharedLibrary' | 'cppSharedLibrary' | 'dotNETAssembly' | 'javaPackage' | 'pythonPackage'

This property is read-only.

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

<b>compiler.build Function</b>	<b>Build Type</b>
<code>compiler.build.productionServerArchive</code>	'productionServerArchive'
<code>compiler.build.comComponent</code>	'comComponent'
<code>compiler.build.cSharedLibrary</code>	'cSharedLibrary'
<code>compiler.build.cppSharedLibrary</code>	'cppSharedLibrary'

<b>compiler.build Function</b>	<b>Build Type</b>
compiler.build.dotNETAssembly	'dotNETAssembly'
compiler.build.javaPackage	'javaPackage'
compiler.build.pythonPackage	'pythonPackage'

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.

<b>Build Type</b>	<b>Files</b>
'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 <code>matlab-data</code> interface: {'path\to\v2\'} {'path\to\GettingStarted.html'} Using the <code>mwArray</code> interface: {'path\to\LibraryName.h'} {'path\to\LibraryName.dll'} {'path\to\LibraryName.lib'} {'path\to\GettingStarted.html'}
'dotNETAssembly'	4×1 cell array {'path\to\AssemblyName.dll'} {'path\to\AssemblyNameNative.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'}

Build Type	Files
'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

**Options – Build options**

ProductionServerArchiveOptions | COMComponentOptions | CSharedLibraryOptions | CppSharedLibraryOptions | DotNETAssemblyOptions | JavaPackageOptions | PythonPackageOptions

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

**Examples**

**Get Build Information from Production Server Archive**

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

Compile using the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

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

results =

Results with properties:

```
BuildType: 'productionServerArchive'
Files: 'D:\Documents\MATLAB\work\magicsquareproductionServerArchive\magicsquare.'
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, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

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

```
results =
```

```
Results with properties:
```

```
BuildType: 'comComponent'  
Files: {2x1 cell}  
Options: [1x1 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, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

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

```
results =
```

```
Results with properties:
```

```
BuildType: 'cSharedLibrary'  
Files: {4x1 cell}  
Options: [1x1 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, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

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

```
results =
```

```
    Results with properties:
```

```
        BuildType: 'cppSharedLibrary'
           Files: {2x1 cell}
           Options: [1x1 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, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

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

```
results =
```

```
    Results with properties:
```

```
        BuildType: 'dotNETAssembly'
           Files: {4x1 cell}
           Options: [1x1 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, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

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

```
results =
```

```
    Results with properties:
```

```
        BuildType: 'javaPackage'
           Files: {3x1 cell}
           Options: [1x1 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, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

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

```
results =
```

```
    Results with properties:
```

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

The Files property contains the paths to the following:

- example folder
- setup.py
- GettingStarted.html

### **See Also**

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

**Introduced in R2020b**



# productionServerCompiler

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

## Syntax

```
productionServerCompiler  
productionServerCompiler project_name
```

## Description

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

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

## Examples

### Create a New Production Server Project

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

```
productionServerCompiler
```

## Input Arguments

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

character array or string

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

## Compatibility Considerations

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

*Not recommended starting in R2020a*

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

**Introduced in R2014a**



# Apps

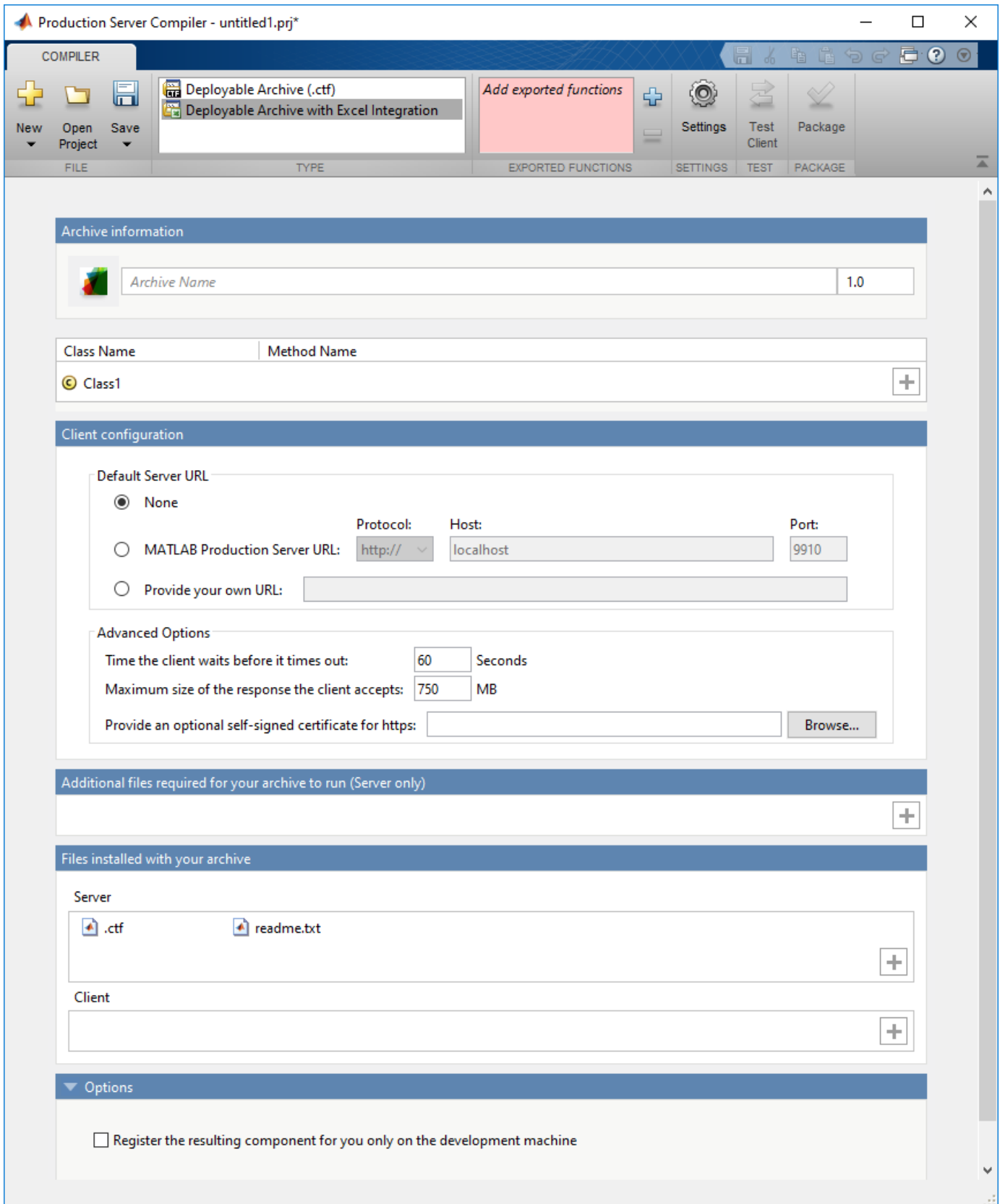
---

## Production Server Compiler

Package MATLAB programs for deployment to MATLAB Production Server

### Description

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



## Open the Production Server Compiler App

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

## Examples

- “Create a deployable archive for MATLAB Production Server”
- “Create and Install a Deployable Archive with Excel Integration For MATLAB Production Server”

## Parameters

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

Deployable Archive | Deployable Archive with Excel Integration

Type of archive to generate as a character array.

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

list of character arrays

Functions to package as a list of character arrays.

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

character array

Name of the archive as a character array.

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

list of files

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

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

list of files

Optional files installed with archive as a list of files.

### **Settings**

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

character array

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

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

character array

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

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

character array

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

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

character array

Folder where generated installers are stored as a character array.

## **Programmatic Use**

productionServerCompiler

## **See Also**

### **Topics**

“Create a deployable archive for MATLAB Production Server”

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

**Introduced in R2013b**





# Client Programming

---

## Create a Java Client Using the MWHttpClient Class

This example shows how to write a MATLAB Production Server client using the Java client API. In your Java code, you will:

- Define a Java interface that represents the MATLAB function.
- Instantiate a 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 called `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.

The interface for the `addmatrix` function

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

```
a = a1 + a2;
```

looks like this:

```
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 “Java Client Programming” (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:

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

These statements initialize the variables used by the application.

- 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 client object's createProxy method 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\_client* 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 client object's close() method 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:

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

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

For example, enter the following:

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

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)

## Create a C# Client Using MWHttpClient

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_client\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` function you compiled in “Package Deployable Archives with Production Server Compiler App” and deployed in “Share Deployable Archive” (MATLAB Production Server).

Create a Python MATLAB Production Server client application:

- 1 Copy the contents of the `matlabroot\toolbox\compiler_sdk\mps_clients\python` folder to your development environment.
- 2 Open a command line,
- 3 Change directories into the folder where you copied the MATLAB Production Server Python client.
- 4 Run the following command.

```
python setup.py install
```

- 5 Start the Python command line interpreter.
- 6 Enter the following import statements at the Python command prompt.

```
import matlab
from production_server import client
```

- 7 Open the connection to the MATLAB Production Server instance and initialize the client runtime.

```
client_obj = client.MWHttpClient("http://localhost:9910")
```

- 8 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]])
```

- 9 Call the deployed MATLAB function.

You must know the following:

- Name of the deployed archive
- Name of the function

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

```
matlab.double([[5.0,7.0,9.0],[9.0,7.0,5.0]])
```

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

- 10 Close the client connection.

```
client_obj.close()
```



## 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 “Share Deployable Archive” (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_client/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\_client* 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.genericErrMsg
<< 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_client/c/include/mps/`.

To link your application, the linker needs access to the following files stored in `matlabroot/toolbox/compiler_sdk/mps_client/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\_client/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
```



# 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')
```





**Introduced in R2018a**

# 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], "mwsizes": [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 following 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 8-3 on the `mps.json.encode` page.

## Output Arguments

### data — MATLAB data

any MATLAB data type

MATLAB data decoded from MATLAB Production Server JSON text returned as the data-type encoded in `text`.

**See Also**

`mps.json.decoderesponse` | `mps.json.encode` | `mps.json.encoderrequest`

**Introduced in R2018a**

# 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

Let `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.encoderrequest` and `mps.json.decoderresponse` 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.

```
data = {struct('name', 'Ed', 'score', 83), struct('name', 'Toni', 'score', 91)};
body = mps.json.encoderrequest(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.decoderresponse(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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `mps.json.encoderrequest(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.encoderrequest(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.

**See Also**

`mps.json.decode` | `mps.json.decoderesponse` | `mps.json.encode`

**Introduced in R2018a**

## 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.encoderesponse` 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.

```
data = {struct('name', 'Ed', 'score', 83), struct('name', 'Toni', 'score', 91)};
body = mps.json.encoderesponse(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.

## See Also

`mps.json.decode` | `mps.json.encode` | `mps.json.encoderesponse`

**Introduced in R2018a**



# 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 a to MATLAB session
stop	Stop a persistence service and detach it from a MATLAB session
restart	Restart a persistence service and attach it to a MATLAB session

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

## Examples

### Create a Redis Service Controller

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

```
ctrl =
```

```
Controller with properties:
```

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

### Create a MATLAB Service Controller

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

```
mctrl =
```

```
Controller with properties:
```

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

## See Also

`mps.cache.DataCache`

### Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

# mps.cache.DataCache

Represent cache concept in MATLAB code

## Description

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

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

## Creation

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

## Properties

See provider-specific subclasses for properties.

## Object Functions

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

## Examples

### Connect to a Redis Cache

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

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

```
c =
```

```
RedisCache with properties:
```

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

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

### **See Also**

`mps.cache.Controller`

### **Topics**

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**



# mps.sync.TimedMATFileMutex

Represent a MAT-file persistence service mutex

## Description

`mps.sync.TimedMATFileMutex` is synchronization primitive used to protect data in a MAT-file database from being simultaneously accessed by multiple workers.

## Creation

Create a `mps.sync.TimedMATFileMutex` object using `mps.sync.mutex`.

## Properties

### Expiration — Duration of lock in seconds

positive integer

This property is read-only.

Duration of advisory lock in seconds.

Example: 10

### ConnectionName — Name of connection

character vector

This property is read-only.

Name of connection to persistence service.

Example: 'myRedisConnection'

### MutexName — Name of lock

character vector

This property is read-only.

Name of advisory lock, specified as a character vector.

Example: 'myMutex'

## Object Functions

<code>mps.sync.mutex</code>	Create a persistence service mutex
<code>acquire</code>	Acquire advisory lock on persistence service mutex
<code>own</code>	Check ownership of advisory lock on a persistence service mutex object
<code>release</code>	Release advisory lock on persistence service mutex

## Examples

### Create a MAT-File Lock Object

```
mctrl = mps.cache.control('myMATFileConnection','MatlabTest','Folder','c:\tmp')
start(mctrl)
lk = mps.sync.mutex('myMATFileMutex','Connection','myMATFileConnection')
```

```
lk =
```

```
TimedMATFileMutex with properties:
```

```
Expiration: 10
ConnectionName: 'myMATFileConnection'
MutexName: 'myMATFileMutex'
```

### See Also

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

### Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

### Introduced in R2018b

# mps.sync.TimedRedisMutex

Represent a Redis persistence service mutex

## Description

`mps.sync.TimedRedisMutex` is a synchronization primitive used to protect data in a Redis persistence service from being simultaneously accessed by multiple workers.

## Creation

Create a `mps.sync.TimedRedisMutex` object using `mps.sync.mutex`.

## Properties

### Expiration — Duration of lock in seconds

positive integer

This property is read-only.

Duration of advisory lock in seconds.

Example: 10

### ConnectionName — Name of connection

character vector

This property is read-only.

Name of connection to persistence service.

Example: 'myRedisConnection'

### MutexName — Name of mutex

character vector

This property is read-only.

Name of mutex, returned as a character vector.

Example: 'myMutex'

## Object Functions

<code>mps.sync.mutex</code>	Create a persistence service mutex
<code>acquire</code>	Acquire advisory lock on persistence service mutex
<code>own</code>	Check ownership of advisory lock on a persistence service mutex object
<code>release</code>	Release advisory lock on persistence service mutex

## Examples

### Create a Redis Lock Object

```
ctrl = mps.cache.control('myRedisConnection', 'Redis', 'Port', 4519);  
start(ctrl)  
lk = mps.sync.mutex('myMutex', 'Connection', 'myRedisConnection')  
  
lk =
```

TimedRedisMutex with properties:

```
Expiration: 10  
ConnectionName: 'myRedisConnection'  
MutexName: 'myMutex'
```

### See Also

[acquire](#) | [mps.sync.TimedMATFileMutex](#) | [mps.sync.mutex](#) | [own](#) | [release](#)

### Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

### Introduced in R2018b

# acquire

Acquire advisory lock on persistence service mutex

## Syntax

```
TF = acquire(lk,timeout)
```

## Description

`TF = acquire(lk,timeout)` acquires an advisory lock and returns a logical 1 (`true`) if the lock was successful, and a logical 0 (`false`) otherwise. If the lock is unavailable, `acquire` will continue trying to acquire it for `timeout` seconds.

## Examples

### Apply Advisory Lock

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

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

Use the connection name to create a persistence service mutex.

```
lk = mps.sync.lock('myDbLock','Connection','myRedisConnection')
```

Try to acquire advisory lock. If lock is unavailable, retry acquiring for 20 seconds.

```
acquire(lk, 20);
```

```
TF =
```

```
    logical
```

```
     1
```

## Input Arguments

### lk — Mutex object

persistence service mutex object

A persistence service specific mutex object. If you use Redis as your persistence provider, `lk` will be a `mps.sync.TimedRedisMutex` object. If you use a MATLAB as your persistence provider, `lk` will be a `mps.sync.TimedMATFileMutex` object.

### timeout — Retry duration

positive integer

Duration after which to retry acquiring lock.

Example: 20

## Output Arguments

### TF — Logical value

logical array

TF has a logical 1 (`true`) if acquiring the advisory lock was successful, and a logical 0 (`false`) otherwise.

### See Also

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

### Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

# attach

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

## Syntax

```
attach(ctrl)
```

## Description

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

## Examples

### Connect a MATLAB Session to a Persistence Service

Attach MATLAB code to a persistence service.

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

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

## Input Arguments

### `ctrl` — Service controller

`mps.cache.Controller` object

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

Example: `attach(ctrl)`

## See Also

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

## Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

## bytes

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

### Syntax

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

### Description

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

### Examples

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

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

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

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

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

b =

```
    72    72    72    80   264
```

tt =

5×2 table

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

### Input Arguments

#### c — Data cache

persistence provider specific data cache object



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

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

Example: `c`

### **keys — Keys**

cell array of character vectors

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

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

## **Output Arguments**

### **b — Number of bytes**

numeric row vector

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

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

## **See Also**

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

## **Topics**

“Use a Data Cache to Persist Data” (MATLAB Production Server)

## **Introduced in R2018b**

## clear

Remove all keys and values from cache

### Syntax

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

### Description

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

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

### Examples

#### Clear All Keys and Values from Cache

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

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

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

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

```
tt =
```

```
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

## See Also

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

## Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

## detach

Disconnect MATLAB session from a persistence service that is already running

### Syntax

```
detach(ctrl)
```

### Description

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

### Examples

#### Disconnect MATLAB Code

Disconnect MATLAB code from a persistence service.

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

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

### Input Arguments

#### **ctrl** — Service controller

`mps.cache.Controller` object

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

Example: `detach(ctrl)`

### See Also

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

### Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

## flush

Write all locally modified keys to the persistence service

### Syntax

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

### Description

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

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

### Examples

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

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

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

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

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

```
tt =
```

```
5x2 table
```

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

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

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

```
c =
```

RedisCache with properties:

```

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

```

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

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

```

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

```

modKeys =

1×1 cell array

```
{'keyOne'}
```

tt =

5×2 table

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

## Input Arguments

### **c** — Data cache

persistence provider specific data cache object

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

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

Example: c

## Output Arguments

### **modKeys** — Modified keys

cell array of character vectors

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

**See Also**

clear | keys | purge | remove | retain

**Topics**

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

## get

Fetch values of keys from cache

### Syntax

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

### Description

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

### Examples

#### Get Values for Keys from Cache

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

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

Add keys and values to the cache.

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

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

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

```
k =
```

```
5×1 cell array
```

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

```
v =
```

```
1×5 cell array
```

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

```
tt =
```



5×2 table

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

## Input Arguments

### **c** — Data cache

persistence provider specific data cache object

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

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

Example: `c`

### **keys** — Keys

cell array of character vectors

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

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

## Output Arguments

### **values** — Values

cell array

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

## See Also

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

### Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

### Introduced in R2018b

## getp

Get the value of a public cache property

### Syntax

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

### Description

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

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

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

### Examples

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

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

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

Retrieve the connection name.

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

```
ans =
```

```
    'myRedisConnection'
```

### Input Arguments

#### **c** — Data cache

persistence provider specific data cache object

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

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

Example: `c`

**property — Property name**

character vector

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

Example: `'Connection'`

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

valid value

A valid property value.

**See Also**

`get` | `keys` | `put`

**Topics**

"Use a Data Cache to Persist Data" (MATLAB Production Server)

**Introduced in R2018b**

## isKey

Determine if the cache contains specified keys

### Syntax

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

### Description

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

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

### Examples

#### Determine if the Cache Contains Specified Keys

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

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

Add keys and values to the cache.

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

Determine if the cache contains specified keys.

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

```
TF =
```

```
1×5 logical array
```

```
1 0 0 0 1
```

### Input Arguments

#### **c** — Data cache

persistence provider specific data cache object

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

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

Example: `c`

**keys — Keys to search for**

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

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

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

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

logical array

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

**See Also**

get | keys | length | put

**Topics**

"Use a Data Cache to Persist Data" (MATLAB Production Server)

**Introduced in R2018b**

## keys

Get all keys from cache

### Syntax

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

### Description

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

### Examples

#### Get Keys from Cache

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

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

Add keys and values to the cache.

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

Get all keys.

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

```
k =
```

```
5×1 cell array
```

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

### Input Arguments

#### **c** — Data cache

persistence provider specific data cache object

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

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

Example: `c`

## Output Arguments

### **k** — Keys

cell array of character vectors

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

## See Also

bytes | get | isKey | length | put

## Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

## length

Number of key-value pairs in the data cache

### Syntax

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

### Description

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

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

### Examples

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

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

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

Retain a few keys locally.

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

Add keys and values to the cache.

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

Count the number of keys-value pairs.

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

```
numTotal =
```

```
int64
```

```
5
```

```
numRemote =
```

```
int64
```

```
3
```



```
numLocal =  
    int64  
    2
```

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

## Input Arguments

### **c** — Data cache

persistence provider specific data cache object

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

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

Example: `c`

### **location** — Location name

'Remote' | 'Local'

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

Example: 'Remote'

## Output Arguments

### **num** — Number of keys

integer

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

## See Also

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

### Topics

"Use a Data Cache to Persist Data" (MATLAB Production Server)

### Introduced in R2018b

## mps.cache.connect

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

### Syntax

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

### Description

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

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

### Examples

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

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

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

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

c =
```

RedisCache with properties:

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

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

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

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

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

```
c =
```

RedisCache with properties:

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

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

## Input Arguments

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

character vector

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

Example: 'myCache'

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

character vector

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

Example: 'Connection', 'myRedisConnection'

## Output Arguments

### **c** — Data cache object

persistence provider-specific data cache object

A persistence provider specific data cache object.

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

## See Also

`mps.cache.DataCache`

### Introduced in R2018b

## mps.cache.control

Create a persistence service controller object

### Syntax

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

### Description

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

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

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

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

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

### Examples

#### Create a Redis Service Controller

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

```
ctrl =
```

Controller with properties:

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

#### Create a MATLAB Service Controller

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

```
mctrl =
```

Controller with properties:

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

```

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

```

## Input Arguments

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

character vector | string

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

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

Example: 'myRedisConnection'

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

'Redis' | 'MatlabTest'

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

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

Example: 'Redis'

Example: 'MatlabTest'

### **num — Port number**

positive scalar

Port number for the persistence service.

Example: 'Port', 4519

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

character vector

Storage folder path, specified as a character vector.

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

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

## Output Arguments

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

mps.cache.Controller object

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

## See Also

mps.cache.Controller | restart | start | stop

### Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

## mps.sync.mutex

Create a persistence service mutex

### Syntax

```
lk = mps.sync.mutex(mutexName, 'Connection', connectionName, Name, Value)
```

### Description

`lk = mps.sync.mutex(mutexName, 'Connection', connectionName, Name, Value)` creates a database advisory lock object.

### Examples

#### Create a Redis Mutex

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

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

Use the connection name to create a persistence service mutex.

```
lk = mps.sync.mutex('myMutex', 'Connection', 'myRedisConnection')
```

```
lk =
```

```
  TimedRedisMutex with properties:
```

```
    Expiration: 10
  ConnectionName: 'myRedisConnection'
    MutexName: 'myMutex'
```

### Input Arguments

#### **mutexName** — Mutex name

character vector

Name of persistence service mutex, specified as a character vector.

Example: 'myMutex'

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

character vector

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

Example: 'Connection', 'myRedisConnection'

### Name-Value Pair Arguments

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

Example: `'Expiration', 10`

### Expiration — Time in seconds

positive integer

Expiration time in seconds after the lock is acquired.

Other clients will be able to acquire the lock even if you do not release it.

Example: `'Expiration', 10`

### Output Arguments

#### `lk` — Mutex object

persistence service mutex object

A persistence service mutex object. If you use Redis as your persistence provider, `lk` will be a `mps.sync.TimedRedisMutex` object. If you use MATLAB as your persistence provider, `lk` will be a `mps.sync.TimedMATFileMutex` object.

### Tips

- A persistence service mutex allows multiple clients to take turns using a shared resource. Each cooperating client creates a mutex object with the same name using a connection to a shared persistence service. To gain exclusive access to the shared resource, a client attempts to acquire a lock on the mutex. When the client finishes operating on the shared resource, it releases the lock. To prevent lockouts should the locking client crash, all locks expire after a certain amount of time.
- Acquiring a lock on a mutex prevents other clients from acquiring a lock on that mutex but it does not lock the persistence service or any keys or values stored in the persistence service. These locks are advisory only and are meant to be used by cooperating clients intent of preventing data corruption. Rogue clients will be able to corrupt or delete data if they do not voluntarily respect the mutex locks.

### See Also

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

### Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

### Introduced in R2018b



## own

Check ownership of advisory lock on a persistence service mutex object

### Syntax

```
TF = own(lk)
```

### Description

`TF = own(lk)` returns a logical 1 (`true`) if you own an advisory lock on the persistence service mutex, and returns a logical 0 (`false`) otherwise.

### Examples

#### Check If You Own the Advisory Lock

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

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

Use the connection name to create a persistence service mutex.

```
lk = mps.sync.lock('myDbLock', 'Connection', 'myRedisConnection')
```

Check if you own the advisory lock.

```
TF = own(lk)
```

```
TF =
```

```
    logical
```

```
    0
```

### Input Arguments

#### lk — Mutex object

persistence service mutex object

A persistence service specific mutex object. If you use Redis as your persistence provider, `lk` will be a `mps.sync.TimedRedisMutex` object. If you use a MATLAB as your persistence provider, `lk` will be a `mps.sync.TimedMATFileMutex` object.

### Output Arguments

#### TF — Logical value

logical array

TF has a logical 1 (`true`) if you own the advisory lock on the persistence service mutex, and a logical 0 (`false`) otherwise.

**See Also**

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

**Topics**

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

# ping

Test whether the persistence service is reachable

## Syntax

```
ping(ctrl)
```

## Description

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

## Examples

### Ping Persistence Service

Test whether the persistence service is reachable.

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

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

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

```
ans =
```

```
    logical
```

```
    1
```

## Input Arguments

### **ctrl** — Service controller

`mps.cache.Controller` object

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

Example: `ping(ctrl)`

## See Also

`restart` | `start` | `stop`

## Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

## purge

Flush all local data to the persistence service

### Syntax

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

### Description

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

### Examples

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

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

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

Add keys and values to the cache.

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

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

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

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

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

```
purgedKeys =
```

```
2×1 cell array
```

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

```
tt =
```

```
5×2 table
```

```
Keys Values
```

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

```
c =
```

```
RedisCache with properties:
```

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

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

## Input Arguments

### **c** — Data cache

persistence provider specific data cache object

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

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

Example: `c`

## Output Arguments

### **purgedKeys** — Purged keys

cell array of character vectors

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

## See Also

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

### Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

# put

Write key-value pairs to cache

## Syntax

```
put(c, key1, value1, ..., keyN, valueN)
put(c, keySet, valueSet)
```

## Description

`put(c, key1, value1, ..., keyN, valueN)` writes key-value pairs to cache. You can store any type of MATLAB data in a cache.

`put(c, keySet, valueSet)` writes key-value pairs to cache with keys from `keySet`, each mapped to a corresponding value from `valueSet`. The input arguments `keySet` and `valueSet` must have the same number of elements, with `keySet` having elements that are unique.

## Examples

### Write Series of Key-Value Pairs to Cache

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

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

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

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

```
tt =
```

```
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 =
```

```
5x2 table
```

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

## Write Object to Cache

Create a class whose object you want to write to the Redis cache.

```
classdef BasicClass
    properties
        Value = pi;
    end
    methods
        function r = roundOff(obj)
            r = round([obj.Value],2);
        end
        function r = multiplyBy(obj,n)
            r = [obj.Value] * n;
        end
    end
end
```

Create an object of the class and assign a value to the Value property,

```
a = BasicClass
a.Value = 4
```

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



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

Add a key and the object that you created to the cache and retrieve the object.

```
put(c,'objKey',a)
objVal = get(c,'objKey')
```

```
objVal =
```

```
    BasicClass with properties:
```

```
    Value: 4
```

The output shows that there is no loss of information during writing an object to the cache and retrieving the object from the cache. The retrieved object contains the same information as the input object.

## Input Arguments

### **c** — Data cache

persistence provider specific data cache object

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

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

Example: `c`

### **key** — Key

character vector

Key to add, specified as a character vector.

Example: `'keyFour'`

### **value** — Value

array

Value, specified as an array. `value` can be any valid MATLAB data type, including MATLAB objects.

Example: `[400, 500]`

### **keySet** — Keys

cell array of character vectors

Keys, specified as a cell array of character vectors.

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

### **valueSet** — Values

cell array

Values, specified as comma-separated cell array. Each value may be any valid MATLAB data type, including MATLAB objects.

Example: `{10, 20, 30, [400 500], magic(5)}`

**See Also**

bytes | clear | get | keys | length | remove

**Topics**

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

# release

Release advisory lock on persistence service mutex

## Syntax

```
TF = release(lk)
```

## Description

`TF = release(lk)` releases an advisory lock on a persistence service mutex. If the lock expires before you release it, `release` returns a logical `0` (`false`). If this occurs, it may indicate potential data corruption.

## Examples

### Release Advisory Lock

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

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

Use the connection name to create a persistence service mutex.

```
lk = mps.sync.lock('myDbLock', 'Connection', 'myRedisConnection')
```

Try to acquire advisory lock. If lock is unavailable, retry acquiring for 20 seconds.

```
acquire(lk, 20);
```

Release lock.

```
TF = release(lk)
```

```
TF =
```

```
    logical
```

```
     1
```

## Input Arguments

### lk — Mutex object

persistence service mutex object

A persistence service specific mutex object. If you use Redis as your persistence provider, `lk` will be a `mps.sync.TimedRedisMutex` object. If you use a MATLAB as your persistence provider, `lk` will be a `mps.sync.TimedMATFileMutex` object.

## Output Arguments

### TF — Logical value

logical array

TF has a logical 1 (`true`) if releasing the advisory lock was successful, and a logical 0 (`false`) otherwise.

### See Also

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

### Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

## remove

Remove keys from cache

### Syntax

```
num = remove(c,keys)
```

### Description

`num = remove(c,keys)` removes keys and associated values from cache. There is no way to recover removed keys.

### Examples

#### Remove Keys from Cache

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

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

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

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

```
tt =
```

```
5×2 table
```

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

Remove two keys from cache `c` and display the remaining keys and values in the cache as a MATLAB table.

```
num = remove(c,{'keyThree','keyFour'})
tt = table(keys(c), get(c,keys(c))','VariableNames',{'Keys','Values'})
```

```
num =
```

```
int64
```

```
2  
  
tt =  
  
3x2 table  
  
Keys          Values  
-----  
'keyFive'    [5x5 double]  
'keyOne'     [          10]  
'keyTwo'     [          20]
```

## Input Arguments

### **c** — Data cache

persistence provider specific data cache object

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

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

Example: `c`

### **keys** — Keys to remove

cell array of character vectors

Keys to remove from cache, specified as a cell array of character vectors.

Example: `{ 'keyThree' , 'keyFour' }`

## Output Arguments

### **num** — Number of keys removed

integer

Number of keys removed, returned as an integer.

## See Also

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

## Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

# restart

Restart a persistence service and attach it to a MATLAB session

## Syntax

```
restart(ctrl)
```

## Description

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

## Examples

### Restart a Persistence Provider

Restart a persistence service.

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

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

## Input Arguments

### `ctrl` — Service controller

`mps.cache.Controller` object

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

Example: `restart(ctrl)`

## See Also

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

## Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

## retain

Store remote keys from cache locally or return locally stored keys

### Syntax

```
retain(c, remoteKeys)
localKeys = retain(c)
```

### Description

`retain(c, remoteKeys)` stores keys from cache locally.

`localKeys = retain(c)` returns a cell array of keys stored locally.

### Examples

#### Store Keys from Cache Locally and Check Local Keys

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

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

Add keys and values to the cache.

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

Retain a few keys locally and check local keys.

```
retain(c, {'keyThree', 'keyFour'})
localKeys = retain(c)
```

```
localKeys =
```

```
    1×2 cell array
```

```
    {'keyThree'}    {'keyFour'}
```

### Input Arguments

#### **c** — Data cache

persistence provider specific data cache object

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

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

Example: c



**remoteKeys — Keys**

cell array of character vectors

Remote keys to store locally, specified as a cell array of character vectors.

Example: { 'keyThree' , 'keyFour' }

**Output Arguments****localKeys — Keys**

cell array of character vectors

Locally stored keys, returned as a cell array of character vectors.

**Tips**

- As a performance optimization you may choose to temporarily store a set of keys and their values in your MATLAB session or worker instead of the persistence service. Keys *retained* in the this fashion will be automatically written to the persistence service (see `flush`) when MATLAB exits or when the first function call returns.
- Manually control the lifetime of retained keys with the `flush` and `purge` methods.

**See Also**

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

**Topics**

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

## start

Start a persistence service and attach it to a MATLAB session

### Syntax

```
start(ctrl)
```

### Description

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

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

### Examples

#### Start a Persistence Service

Start a persistence service.

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

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

### Input Arguments

#### `ctrl` — Service controller

`mps.cache.Controller` object

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

Example: `start(ctrl)`

### See Also

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

#### Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

## stop

Stop a persistence service and detach it from a MATLAB session

### Syntax

```
stop(ctrl)
```

### Description

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

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

### Examples

#### Stop a Persistence Service

Stop a persistence service.

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

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

### Input Arguments

#### **ctrl** — Service controller

`mps.cache.Controller` object

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

Example: `stop(ctrl)`

### See Also

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

#### Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

## version

Version number for persistence provider

### Syntax

```
version(ctrl)
```

### Description

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

### Examples

#### Get Version Number

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

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

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

```
Redis version: 3.0.504
```

### Input Arguments

#### **ctrl** — Service controller

`mps.cache.Controller` object

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

Example: `version(ctrl)`

### See Also

`restart` | `start` | `stop`

### Topics

“Use a Data Cache to Persist Data” (MATLAB Production Server)

**Introduced in R2018b**

