# MATLAB® Production Server™

Getting Started

MATLAB®

**R**2024**a**

MathWorks®

## How to Contact MathWorks

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

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

*MATLAB® Production Server™ Getting Started Guide*

**Revision History**

| | | |
|---|---|---|
| September 2012 | Online only | New for Version 1.0 (Release R2012b) |
| March 2013 | Online only | Revised for Version 1.0.1 (Release R2013a) |
| October 2013 | Online only | Revised for Version 1.1 (Release R2013b) |
| March 2014 | Online only | Revised for Version 1.2 (Release R2014a) |
| October 2014 | Online only | Revised for Version 2.0 (Release R2014b) |
| March 2015 | Online only | Revised for Version 2.1 (Release R2015a) |
| September 2015 | Online only | Revised for Version 2.2 (Release R2015b) |
| March 2016 | Online only | Revised for Version 2.3 (Release 2016a) |
| September 2016 | Online only | Revised for Version 2.4 (Release 2016b) |
| March 2017 | Online only | Revised for Version 3.0 (Release 2017a) |
| September 2017 | Online only | Revised for Version 3.0.1 (Release R2017b) |
| March 2018 | Online only | Revised for Version 3.1 (Release R2018a) |
| September 2018 | Online only | Revised for Version 4.0 (Release R2018b) |
| March 2019 | Online only | Revised for Version 4.1 (Release R2019a) |
| September 2019 | Online only | Revised for Version 4.2 (Release R2019b) |
| March 2020 | Online only | Revised for Version 4.3 (Release R2020a) |
| September 2020 | Online only | Revised for Version 4.4 (Release R2020b) |
| March 2021 | Online only | Revised for Version 4.5 (Release R2021a) |
| September 2021 | Online only | Revised for Version 4.6 (Release R2021b) |
| March 2022 | Online only | Revised for Version 5.0 (Release R2022a) |
| September 2022 | Online only | Revised for Version 5.1 (Release R2022b) |
| March 2023 | Online only | Revised for Version 5.2 (Release R2023a) |
| September 2023 | Online only | Revised for Version 23.2 (R2023b) |
| March 2024 | Online only | Revised for Version 24.1 (R2024a) |

# Contents

# 4 Licensing

# 5 Deploying an Application

# Overview

## MATLAB Production Server Product Description

**Integrate MATLAB algorithms into web, database, and enterprise applications**

MATLAB Production Server lets you incorporate custom analytics into web, database, and production enterprise applications running on dedicated servers or in the cloud. You can create algorithms in MATLAB, package them using MATLAB Compiler SDK™, and then deploy them to MATLAB Production Server without recoding or creating custom infrastructure. Users can then access the latest version of your analytics automatically. Each algorithm, when deployed, can behave like a MATLAB function or as a web request handler.

MATLAB Production Server manages multiple MATLAB Runtime versions simultaneously. As a result, algorithms developed in different versions of MATLAB can be incorporated into your application. The server runs on multiprocessor and multicore computers, providing low-latency processing of concurrent work requests. You can deploy the server on additional computing nodes to scale capacity and provide redundancy.

# MATLAB Production Server Workflow

The following figure illustrates the basic workflow to deploy MATLAB code using MATLAB Production Server.

## Create Deployable Archives

MATLAB application developers write MATLAB functions, and compile them into deployable archives using MATLAB Compiler SDK. For more information, see “Create Deployable Archive for MATLAB Production Server”.

## Deploy Archives to MATLAB Production Server

You can run MATLAB Production Server instances on-premises, in the cloud, or in Kubernetes®.

- Before you can deploy an archive to an on-premises instance, you need to create and configure a server instance. Configuring a server instance includes tasks such as configuring licenses and installing MATLAB Runtime. You can use the command line or the dashboard to manage the server. For more information, see “Set Up MATLAB Production Server Using the Command Line” on page 5-7 and “Set Up and Log In to MATLAB Production Server Dashboard”.
- Cloud deployments are pre-configured to use MATLAB Runtime. Cloud deployments offer the option of using a MATLAB Production Server license or pay-as-you-go for the services you use, where you do not require a license. For more information about MATLAB Production Server cloud offerings, see “Cloud Deployment”.
- To configure a server in Kubernetes, you need to pull MATLAB Runtime container images from the MathWorks® container registry and configure licensing. For deployment details in Kubernetes, see the MATLAB Production Server in Kubernetes repository on GitHub®.

After a server is set up, server administrators can deploy the archives into one or more instances of the MATLAB Production Server. For more information, see “Deploy Archive to MATLAB Production Server” on page 5-9.

## Write Client applications to Invoke Deployed MATLAB Code

Application developers use MATLAB Production Server client APIs or the MATLAB Production Server RESTful API to invoke MATLAB code deployed on the server. You can write client code in .NET, Java®, Python®, C, MATLAB, and web-based languages such as JavaScript®. For more information, see “Client Programming”.

If applicable, you can use application installers to distribute and install client applications on end-user computers.

## See Also

## More About

- “Set Up MATLAB Production Server Using the Command Line” on page 5-7
- “Deploy Archive to MATLAB Production Server” on page 5-9

- “Client Programming”
- “Server Overview”

# Installation

## Install MATLAB Production Server Product

**Note**

1. Prior to installing any MathWorks products, review instructions in “Install Products on Client Machines”.
2. MATLAB Production Server requires a network license manager. For details, see “Install License Manager on License Server”.

   If your organization already has a network license manager installed, add the MATLAB Production Server license file to it.

### Video Walkthrough on Windows

For a visual walkthrough of the installation and setup on Windows®, watch the video.

### Install

**Note** MATLAB Production Server is not supported on macOS (Apple silicon).

Installing the MATLAB Production Server product requires a valid software license, which you can obtain by purchasing the product or requesting a trial. For more information, visit the MathWorks Store.

To start the installation, run the MathWorks installer and select the MATLAB Production Server product for installation. To download the installer, visit MathWorks Downloads. For instructions, see “Download and Install MATLAB”. For installation instructions without an internet connection, see “Install MathWorks Products on Offline Computer”.

The installation process creates a default installation folder based on your operating system and the release version. For instance, the default installation folder for R2024a is as follows:

| **Operating System** | **Default Installation Folder** |
|---|---|
| Windows | `C:\Program Files\MATLAB\MATLAB Production Server\R2024a` |
| Linux® | `/usr/local/MATLAB/MATLAB_Production_Server/R2024a` |
| macOS (Intel® processor) | `/Applications/MATLAB/MATLAB_Production_Server/R2024a` |

Once the installation is complete, you can set up and configure the server from the *`$MPS_INSTALL`*`\script` folder. For details, see “Set Up MATLAB Production Server Using the Command Line” on page 5-7.

To configure licenses for use on cloud platforms, see “Configure MATLAB Production Server License for Use on the Cloud”.

## Uninstall

To uninstall the MATLAB Production Server product, follow the usual instructions for uninstalling MathWorks products. For more information, see “Uninstall MATLAB”.

## See Also

`mps-start` | `mps-setup`

## More About

- “Set Up MATLAB Production Server Using the Command Line” on page 5-7
- “Install Products on Client Machines”

## Supported MATLAB Runtime Versions for MATLAB Production Server

MATLAB Runtime is a standalone set of shared libraries that enable the execution of compiled MATLAB applications or components on computers that do not have MATLAB installed. MATLAB Production Server requires a MATLAB Runtime instance to execute the deployed MATLAB applications that it hosts. If your on-premises server machine does not have MATLAB Runtime installed, download and install the MATLAB Runtime from `https://www.mathworks.com/products/compiler/mcr`. For a server deployment on the cloud, the deployment provides the supported MATLAB Runtime versions. For more information about MATLAB Runtime, see "MATLAB Runtime" (MATLAB Compiler).

**Note** An installation of MATLAB Production Server supports MATLAB Runtime versions up to six releases back.

The following table lists several MATLAB Production Server releases and the corresponding MATLAB Runtime versions that each release supports.

| MATLAB Production Server Release | Supported MATLAB Runtime Versions |
|---|---|
| R2023b | R2023b, R2023a, R2022b, R2022a, R2021b, R2021a |
| R2023a | R2023a, R2022b, R2022a, R2021b, R2021a, R2020b |
| R2022b | R2022b, R2022a, R2021b, R2021a, R2020b, R2020a |
| R2022a | R2022a, R2021b, R2021a, R2020b, R2020a, R2019b |
| R2021b | R2021b, R2021a, R2020b, R2020a, R2019b, R20219a |
| R2021a | R2021a, R2020b, R2020a, R2019b, R20219a, R2018b |

To set the MATLAB Runtime version of a server instance, use the `mps-setup` command or the `mcr-root` configuration property.

### See Also

### More About

- "Specify Default MATLAB Runtime for New Server Instances" on page 3-4
- "Support Multiple MATLAB Runtime Versions"

# Set Up

## Create Server Instance Using Command Line

Before you can deploy MATLAB code with MATLAB Production Server, you must create a server instance to host your deployable archive. A server instance is one unique configuration of the MATLAB Production Server product. Each configuration has its own parameter settings file (`main_config`), as well as its own set of diagnostic files.

### Prerequisites

Before creating a server instance using the command line, ensure you have:

- installed MATLAB Production Server on your machine. For more information, see "Install MATLAB Production Server Product" on page 2-2.
- added the `script` folder to your system `PATH` environment variable. Doing so enables you to run server commands such as `mps-new` from any folder on your system.

You can also run server commands from the *`$MPS_INSTALL`*`\script` folder, where *`$MPS_INSTALL`* is the location where MATLAB Production Server is installed. For example, on Windows, the default location is `C:\Program Files\MATLAB\MATLAB Production Server\`*`ver`*`\script`. *`ver`* is the version of MATLAB Production Server.

### Procedure

To create a server instance from the command line, enter the `mps-new` command from the system prompt. Specify the name of the server that you want to create as an argument to the `mps-new` command.

```
mps-new [path\]server_name [-v]
```

- *`path`* — path to the server instance and configuration that you want to create for use with the MATLAB Production Server product.

  If you want to create a server instance in the current folder, you do not need to specify a full path; only specify the server name.
- *`server_name`* — name of the server instance and configuration that you want to create.
- `-v` — enable verbose output to get the information and status about each folder created in the server configuration.

For example, to create a server instance with the name `prod_server_1` located in `C:\tmp` and use the verbose mode, run the following on your system command prompt.

```
C:\tmp>mps-new prod_server_1 -v
```

The command generates the following output.

```
prod_server_1\.mps_version...ok
prod_server_1\config\main_config...ok
prod_server_1\.mps_socket...ok
prod_server_1\auto_deploy...ok
prod_server_1\endpoint...ok
prod_server_1\log...ok
prod_server_1\old_logs...ok
prod_server_1\pid...ok
```

```
prod_server_1\x509...ok
The UUID of the newly created instance is 4876f876-56a6-40ef-a4e3-96a69b39cb49
```

For more information on the folders created in a server configuration, see “Server Diagnostic Tools”.

For creating server instances on the cloud, see “Cloud Deployment”.

## See Also

`mps-service` | `mps-new`

## More About

- “Configure Server Instance as Windows Service”
- “Start Server Instance Using Command Line” on page 3-6

## Specify Default MATLAB Runtime for New Server Instances

Each server instance that you create with MATLAB Production Server has its own configuration file that defines various server management criteria. Use the `mps-setup` command to set the default MATLAB Runtime for all on-premises server instances that you create. The `mps-setup` command line wizard searches your machine for installed MATLAB Runtime instances and sets the default path to the MATLAB Runtime for all server instances.

If you do not have MATLAB Runtime installed on your machine, you must install it first. For more information, see "Supported MATLAB Runtime Versions for MATLAB Production Server" on page 2-4.

To set the default MATLAB Runtime:

1. Open a system command prompt with `administrator` privileges.
2. From the command prompt, navigate to the MATLAB Production Server `script` folder and run `mps-setup`.

   Alternatively, add the `script` folder to your system `PATH` environment variable to run `mps-setup` from any folder on your system. The `script` folder is located at *`$MPS_INSTALL`*`\script`, where *`$MPS_INSTALL`* is the location in which MATLAB Production Server is installed. For example, on Windows, the default location for the `script` folder is `C:\Program Files\MATLAB\MATLAB Production Server\`*`ver`*`\script\mps-setup`, where *`ver`* is the version of MATLAB Production Server.
3. Follow the instructions in the command line wizard.

   The wizard searches your system for installed MATLAB Runtime instances and displays them.
4. Enter `y` to confirm or `n` to specify a default MATLAB Runtime for all server instances.

   If `mps-setup` cannot locate an installed MATLAB Runtime on your system, the wizard prompts you to enter a path name to a valid instance.

### Run mps-setup in Non-Interactive Mode for Silent Install

You can also run `mps-setup` without interactive command input for silent installations. To do so, specify the path name of the MATLAB Runtime as a command line argument.

For example, on Windows, run the following at the system command prompt.

```
C:\>mps-setup "C:\Program Files\MATLAB\MATLAB Runtime\mcrver"
```

*`mcrver`* is the version of the MATLAB Runtime to use.

### Update MATLAB Runtime After Server Setup

If the server is already set up and you need to change or add a MATLAB Runtime version to the server, in the `config/main_config` file of your server instance root, update the `mcr-root` property.

### See Also

`mps-setup`

## More About

- “Specify MATLAB Runtime for Server Instance Using Command Line”
- “Support Multiple MATLAB Runtime Versions”

## Start Server Instance Using Command Line

| **In this section...** |
|---|
| "Prerequisites" on page 3-6 |
| "Procedure" on page 3-6 |

Follow the procedure below to start a server instance from the command line in an on-premises installation of MATLAB Production Server.

### Prerequisites

Before attempting to start a server, verify that you have:

- Installed the MATLAB Runtime. For more information, see "Supported MATLAB Runtime Versions for MATLAB Production Server" on page 2-4.
- Specified the default MATLAB Runtime for the server instance. For more information, see "Specify Default MATLAB Runtime for New Server Instances" on page 3-4.
- Created a server instance. For more information, see "Create Server Instance Using Command Line" on page 3-2.

### Procedure

To start a server instance, complete the following steps:

1. Open a system command prompt.
2. Enter the `mps-start` command:

   ```
   mps-start [-C path/]server_name [-f]
   ```

   where:

   - `-C path/` — Path to the server instance that you want to start. *`path`* should end with the server name.
   - *`server_name`* — Name of the server instance you want to start.
   - `-f` — Force command to succeed, regardless of whether the server is already started or stopped.

After you start a server, you can verify that the server is running by using the `mps-status` command.

### See Also

`mps-start` | `mps-service` | `mps-new` | `mps-stop`

### More About

- "Configure Server Instance as Windows Service"
- "Create Server Instance Using Command Line" on page 3-2
- "Deploy Archive to MATLAB Production Server" on page 5-9

# Verify Server Status

**In this section...**

Use the `mps-status` command to verify the status of an on-premises MATLAB Production Server instance. You can use `mps-status` after issuing commands such as `mps-start`, `mps-stop`, and `mps-restart` to check if the server has started or stopped. The output of `mps-status` contains information about the status of the MATLAB Production Server instance and also information about the license server that the MATLAB Production Server instance uses.

## Procedure

1 Open a system command prompt.
2 Enter the following command:

```
mps-status [-C path/]server_name
```

where:

- `-C path/` — Path to the server instance. *`path`* should end with the name of the server to be queried for status.
- *`server_name`* — Name of the server to be queried for status.

### Example

To verify the status of a server instance `prod_server_1` located at `\tmp\prod_server_1`, type the following at the system command prompt:

```
mps-status -C \tmp\prod_server_1
```

Depending on whether the server is able to check out a valid license, `mps-status` returns different responses.

- If `prod_server_1` is running and operating with a valid license, it returns the following response.

```
\tmp\prod_server_1 STARTED
License checked out
```

- If `prod_server_1` is unable to check out a valid license, the server returns either a warning when it is within the license checkout grace period or returns an error if it is past the grace period. For details, see `license-grace-period`.

```
\tmp\prod_server_1 STARTED
WARNING: lost connection to license server -
request processing will be disabled at 2019-Jun-27
15:40:31.002137 Eastern Daylight Time unless
connection to license server is restored.
```

or

```
\tmp\prod_server_1 STARTED
ERROR: lost connection to license server -
request processing disabled.
```

## License Server Status Information

In addition to the status of the MATLAB Production Server instance, `mps-status` also displays the status of the license server associated with the server you are querying. The following table lists different license server status messages from the output of `mps-status`.

| License Server Status Message | Message Description |
|---|---|
| `License checked out` | The server is operating with a valid license. The server is communicating with the License Manager, and is able to check out the required number of license keys. |
| `WARNING: lost connection to license server - request processing will be disabled at` *`time`* `unless connection to license server is restored` | The server has lost communication with the License Manager, but the server is still fully operational and will remain operational until the specified *`time`*. At *`time`*, if the server is still unable to connect to the license server, the server will be unable to process requests until licensing is reestablished. |
| `ERROR: lost connection to license server - request processing disabled` | The server has lost communication with the License Manager for a period of time exceeding the grace period. See `license-grace-period`. Request processing is suspended, but the server is actively attempting to reestablish communication with the License Manager. Request processing resumes if the sever is able to reestablish communication with the License Manager. |

### See Also
`mps-status` | `mps-stop` | `mps-restart` | `mps-start`

### More About
- "Health Check API"

# Licensing

## Manage Licenses for MATLAB Production Server

The Network License Manager manages MATLAB Production Server licenses. For more information, see “Install Products on Client Machines”.

In addition to following instructions in the License Center to obtain and activate your license, you must configure MATLAB Production Server to use the license files.

### Specify or Verify License Server Options in Server Configuration File

After you create a server from the command line using the `mps-new` command, you can specify or verify values for License Server properties in the `main_config` server configuration file. To update values for License Server properties, open *`server_name`*`/config/main_config` in a text editor.

- `license` — Specifies license servers and license files. You can specify multiple license servers including port numbers (*`port_number@license_server_name`*), as well as license files, with one entry in `main_config`. List where you want the product to search, in order of precedence, using semi-colons (;) as separators on Windows or colons (:) as separators on Linux.

  For example, on a Linux system, you can specify the following value for the `license` property:

  ```
  --license 27000@hostA:/opt/license/license.dat:27001@hostB:./license.dat
  ```

  The system searches the property values in the following order:

  1. `27000@hostA:` (`hostA` configured on port `27000`)
  2. `/opt/license/license.dat` (local license data file)
  3. `27001@hostB:` (`hostB` configured on port `27001`)
  4. `./license.dat` (local license data file)
- `license-grace-period` — The maximum length of time that MATLAB Production Server responds to HTTP requests, after license server heartbeat has been lost. See the network license manager documentation for more on heartbeats and related license terminology.
- `license-poll-interval` — The interval of time that must pass after license server heartbeat has been lost and MATLAB Production Server stops responding to HTTP requests, before the system polls the license server to verify and checkout a valid license. Polling occurs at the interval specified by `license-poll-interval` until the server is able to check out a license. See the network license manager documentation for more on heartbeats and related license terminology.

### Verify Status of License Server Using mps-status

The `mps-status` command returns the status of the server instance and the associated license server. The command is available only in an on-premises installation of MATLAB Production Server.

For detailed descriptions of these status messages, see “License Server Status Information” on page 3-8.

### Forcing a License Checkout Using mps-license-reset

Use the `mps-license-reset` command to force the server to checkout a license in an on-premises MATLAB Production Server installation. You can use this command at any time, if you do not want to wait for MATLAB Production Server to verify and checkout a license at an interval established by a server configuration option such as `license-grace-period` or `license-poll-interval`.

## See Also

`mps-status` | `mps-license-reset`

## Related Examples

- “Health Check API”
- “Install Products on Client Machines”

# Deploying an Application

## Create Deployable Archive for MATLAB Production Server

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

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

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

### Create MATLAB Function

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

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

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

a = a1 + a2;
```

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

The output is:

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

### Create Deployable Archive with Production Server Compiler App

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

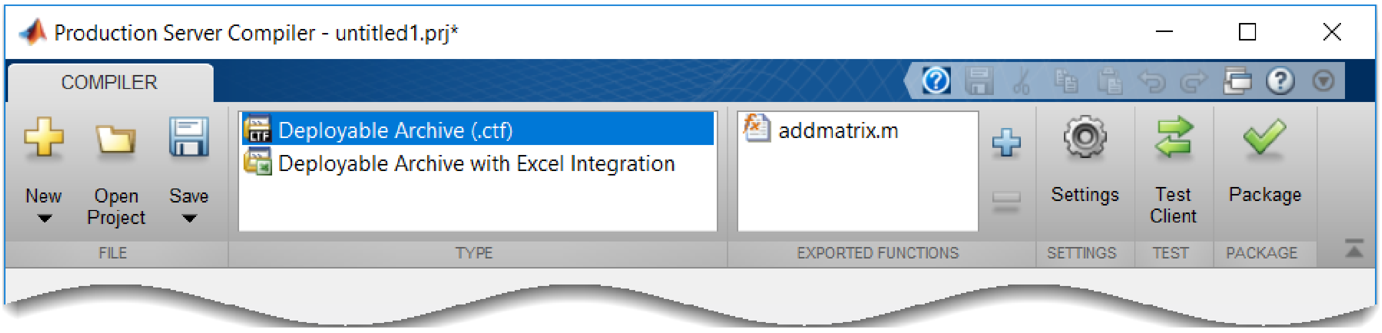
1. To open the **Production Server Compiler** app, type `productionServerCompiler` at the MATLAB prompt.

   Alternatively, on the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Production Server Compiler**. In the **Production Server Compiler** project window, click **Deployable Archive (.ctf)**.
2. In the **Production Server Compiler** project window, specify the main file of the MATLAB application that you want to deploy.

   1. In the **Exported Functions** section, click ➕.
   2. In the **Add Files** window, browse to the example folder, and select the function you want to package.

      Click **Open**.

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


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

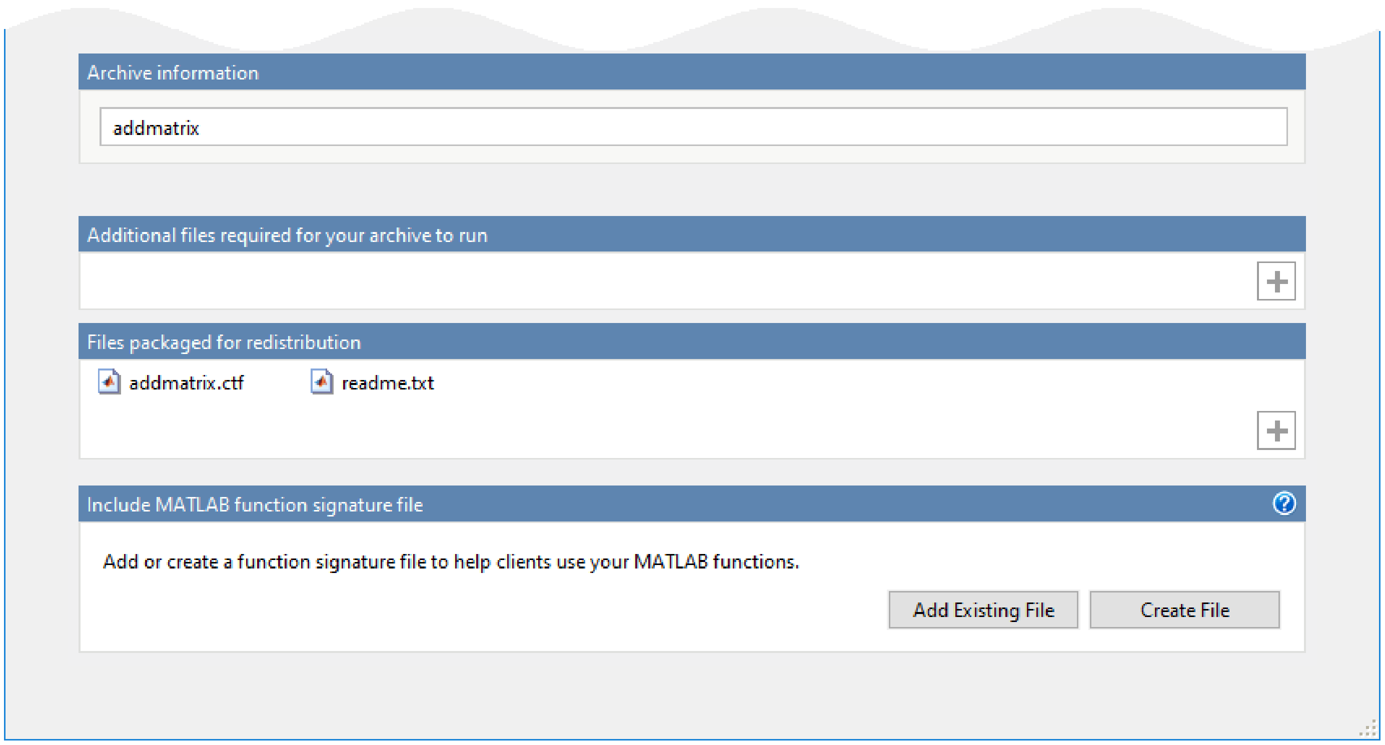
## Customize Application and Its Appearance

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

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

  - Generated deployable archive
  - Generated `readme.txt`

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


Archive information

addmatrix

Additional files required for your archive to run

Files packaged for redistribution

addmatrix.ctf readme.txt

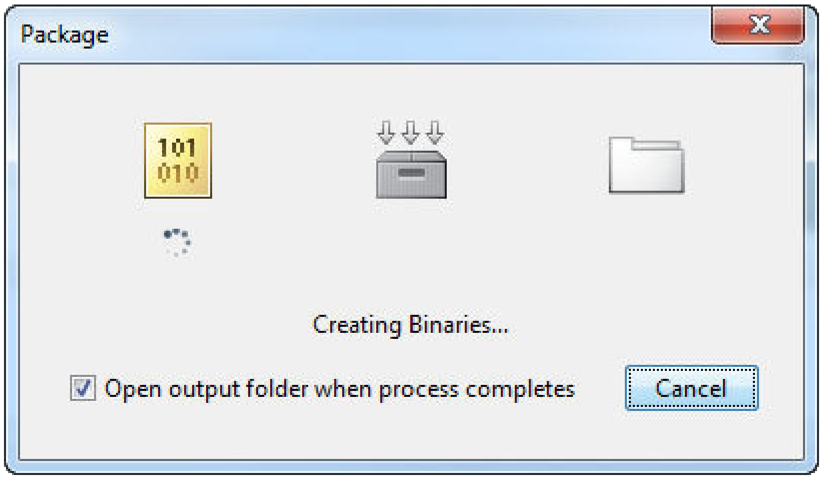
Include MATLAB function signature file

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

## Package Application

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

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


Package

Creating Binaries...

☑ Open output folder when process completes

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

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

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

- `PackagingLog.html` — Log file generated by MATLAB Compiler SDK

## Create Deployable Archive Using compiler.build.productionServerArchive

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

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

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

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

  ```
  buildResults = 

    Results with properties:

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

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

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

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

  - `addmatrix.ctf` — Deployable archive file.
  - `includedSupportPackages.txt` — Text file that lists all support files included in the assembly.
  - `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations (MATLAB Compiler).
  - `readme.txt` — Text file that contains packaging and deployment information.
  - `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
  - `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

## Compatibility Considerations

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

## See Also

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

**Apps**
Production Server Compiler

## More About

- “Test Client Data Integration Against MATLAB” (MATLAB Compiler SDK)
- “Deploy Archive to MATLAB Production Server” on page 5-9
- “MATLAB Function Signatures in JSON”
- “JSON Representation of MATLAB Data Types”

## Set Up MATLAB Production Server Using the Command Line

You can run MATLAB Production Server on-premises and in the cloud. To set up an on-premises server, you can use the system command line or the dashboard, which is a web-based interface. The following procedure use the command line. For information on interacting with the server using the dashboard, see "Server Management Using Dashboard". For using MATLAB Production Server on the cloud, see "Cloud Deployment".

**Note** MATLAB functions deployed to the server are created using MATLAB Compiler SDK. For details on how to create deployable archives for deployment to the server, see "Enterprise Deployment with MATLAB Production Server" (MATLAB Compiler SDK).

### Prerequisites

- Verify that you have installed the MATLAB Production Server product. For details, see "Install MATLAB Production Server Product" on page 2-2.
- Verify that you have installed a network license manager. For details, see "Install License Manager on License Server".
- Verify that you have MATLAB Runtime installed. For details, see "Install and Configure MATLAB Runtime". For supported versions, see "Supported MATLAB Runtime Versions for MATLAB Production Server" on page 2-4.

### Set Up the Server

1 After installing the MATLAB Production Server product, navigate to the folder containing the MATLAB Production Server command-line scripts.

| Operating System | Default Location of Command-Line Scripts |
|---|---|
| Windows *(Administrator)* | `C:\Program Files\MATLAB\MATLAB Production Server\R2024a\script` |
| Linux *(sudo)* | `/usr/local/MATLAB/MATLAB_Production_Server/R2024a/script` |
| macOS *(sudo)* | `/Applications/MATLAB/MATLAB_Production_Server/R2024a/script` |

2 At the operating system command line, start the interactive setup interface by typing:

```
mps-setup
```

For details, see `mps-setup`.

During the setup process set the default MATLAB Runtime version you want the server to use. For details, see "Specify Default MATLAB Runtime for New Server Instances" on page 3-4.

## Create Server Instance

To create a server configuration or instance, enter the `mps-new` command from the system prompt. Specify the name of the server that you want to create as an argument to the `mps-new` command.

For example, to create a server instance with the name `prod_server_1` located in `C:\tmp` and to use the verbose mode, run the following on your system command prompt.

```
C:\tmp>mps-new prod_server_1 -v
```

For more information, see “Create Server Instance Using Command Line”.

## Configure Server Instance

After you create a new server instance, you must configure it. At a minimum, ensure that the following properties in the `main_config` server configuration file are set.

- license — Specify the host and port of the license server, typically `27000@license-server-host`.
- mcr-root — Specify the path to the versions of MATLAB Runtime to use. You can set this property to use multiple versions of MATLAB Runtime. For more information, see “Support Multiple MATLAB Runtime Versions”.

For more information on editing `main_config`, see Server Configuration Properties.

## Start Server Instance

To start the server instance that you created, enter the `mps-start` command from the system prompt. Specify the name of the server that you want to start as an argument to the `mps-start` command.

For example, to start a server instance with the name `prod_server_1` located in `C:\tmp`, run the following on your system command prompt.

```
C:\tmp>mps-start -C prod_server_1
```

For more information, see “Start Server Instance Using Command Line”.

## See Also
`mps-start` | `mps-setup` | `mps-status` | `mps-restart`

## More About

- “Install MATLAB Production Server Product” on page 2-2
- “Manage Licenses for MATLAB Production Server” on page 4-2
- “Enable HTTPS”

## Deploy Archive to MATLAB Production Server

After you package MATLAB functions into MATLAB Production Server deployable archives, you can upload the archives to your server instance. For clients to access the archives, your server instance must be running.

The deployable archive has the name `*project_name*.ctf`. If you use the Production Server Compiler app to create a deployable archive, the archive is available in the `for_redistribution` folder of the deployment project. If you use the `compiler.build.productionServerArchive` function to create a deployable archive, you can specify an output folder to create the deployable archive.

For on-premises server instances managed using the command line, copy the deployable archive into the folder specified by the auto-deploy-root property. By default, the server uses the `auto_deploy` folder in your server instance. You can add a deployable archive into the `auto_deploy` folder of a running server. The server monitors this folder dynamically and processes the deployable archives that are added to the `auto_deploy` folder.

For on-premises server instances managed using the dashboard and server deployments on Azure® and AWS®, use the dashboard and cloud dashboard, respectively, to upload and share MATLAB applications. For uploading applications for server deployments in Azure, see “Upload MATLAB Application” and “Upload MATLAB Applications”. For uploading applications for server deployments in AWS, see “Upload MATLAB Application” and “Upload MATLAB Applications”.

For servers running in Kubernetes, upload the deployable archive to your network file server or Azure file share. All users must have read permission to the deployable archive. For deployment details in Kubernetes, see the MATLAB Production Server in Kubernetes repository on GitHub.

### See Also

**Apps**
Production Server Compiler

**Functions**
`mcc`

### More About

- “Create Deployable Archive for MATLAB Production Server”
- “Create Server Instance Using Command Line”
- “Start Server Instance Using Dashboard”

## Create MATLAB Production Server Java Client Using MWHttpClient Class

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

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

To create a Java MATLAB Production Server client application:

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

   ```
   import java.net.URL;
   import java.io.IOException;
   import com.mathworks.mps.client.MWClient;
   import com.mathworks.mps.client.MWHttpClient;
   import com.mathworks.mps.client.MATLABException;
   ```
4. Add a Java interface that represents the deployed MATLAB function.

   For example, consider the following `addmatrix` function deployed to the server. For information on writing and compiling the function for deployment, see “Create Deployable Archive for MATLAB Production Server”. For deploying the function to the server, see “Deploy Archive to MATLAB Production Server” on page 5-9.

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

   a = a1 + a2;
   ```

   The interface for the `addmatrix` function follows.

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

   When creating the interface, note the following:

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

```
public class MPSClientExample
{
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

For more information about the `createProxy` method, see the Javadoc included in the *`$MPS_INSTALL`*`/client` folder, where *`$MPS_INSTALL`* is the name of your MATLAB Production Server installation folder.

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

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

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

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

12 Save the Java file.

The completed Java file should resemble the following:

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

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

```
public class MPSClientExample {

    public static void main(String[] args){

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

        MWClient client = new MWHttpClient();

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

            // Print the resulting matrix
            printResult(result);

        }catch(MATLABException ex){

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

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

            client.close();
        }
    }

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

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

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

```
javac -classpath "MPS_INSTALL_ROOT\client\java\mps_client.jar" MPSClientExample.java
```

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

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

```
java -classpath .;"MPS_INSTALL_ROOT\client\java\mps_client.jar" MPSClientExample
```

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

   The application returns the following at the console:

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

## See Also

## More About

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

- “Asynchronous RESTful Requests Using Protocol Buffers in the Java Client”

## Create a C# Client

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

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

### Create Microsoft Visual Studio Project

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

### Create Reference to Client Runtime Library

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

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

   In an on-premises MATLAB Production Server installation, the library is located in *`$MPS_INSTALL`*`\client\dotnet`, where *`$MPS_INSTALL`* is the location in which MATLAB Production Server is installed. 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);
```

For information on creating and deploying an archive to the server, see “Create Deployable Archive for MATLAB Production Server” and “Deploy Archive to MATLAB Production Server” on page 5-9.

### 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”
- “Configure the Client-Server Connection”
- “Synchronous RESTful Requests Using Protocol Buffers in .NET Client”

# 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” (MATLAB Compiler SDK) and deployed in “Deploy Archive to MATLAB Production Server” on page 5-9.

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 *`$MPS_INSTALL`*`/client/c/include/mps` folder where *`$MPS_INSTALL`* is the root folder which MATLAB Production Server is installed.
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 *`$MPS_INSTALL`*`/client` folder, where *`$MPS_INSTALL`* is the name of your MATLAB Production Server installation folder.

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

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

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

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

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

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

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

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

**14** Free the memory used by the inputs.

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

**15** Free the memory used by the outputs.

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

**16** Free the memory used by the client runtime.

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

**17** Save the file.

The completed program should resemble the following:

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

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

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

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

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

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

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

  status = mpsruntime->feval(context,
             "http://localhost:9910/addmatrix/addmatrix",
             numOut, outVal, numIn, (const mpsArray **)inVal);
    if (status==MPS_OK)
    {
      double* out = mpsGetPr(outVal[0]);

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

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

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

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

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

**18** Compile the application.

To compile your client code, the compiler needs access to `client.h`. This header file is stored in `$MPSROOT/client/c/include/mps/`.

To link your application, the linker needs access to the following files stored in `$MPSROOT/client/c/<arch>/lib/`:

**Files Required for Linking**

| Windows | UNIX®/Linux | Mac OS X |
|---|---|---|
| `$arch\lib\mpsclient.lib` | `$arch/lib/libprotobuf3.so` | `$arch/lib/libprotobuf3.dylib` |
| | `$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 `$MPSROOT/client/c/<arch>/lib/` to the application's path:

**Files Required for Running**

| Windows | UNIX/Linux | Mac OS X |
|---|---|---|
| `$arch\lib\mpsclient.dll` | `$arch/lib/libprotobuf3.so` | `$arch/lib/libprotobuf3.dylib` |
| `$arch\lib\libprotobuf3.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
```

## Create a Python Client

This example shows how to write a MATLAB Production Server client using the Python client API. The client application calls the `addmatrix` MATLAB function deployed to a server instance. For information on writing and compiling the function for deployment, see “Create Deployable Archive for MATLAB Production Server”. For deploying the function to the server, see “Deploy Archive to MATLAB Production Server” on page 5-9.

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

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

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

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

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

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

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

   The output is:

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

   ```
   client_obj.close()
   ```

### See Also
`matlab.production_server.client.MWHttpClient`

### Related Examples
- “Create Client Connection”
- “Invoke Packaged MATLAB Functions”

## Isolate Deployable Archives in MATLAB Production Server

The `component-isolation` property in the `main_config` file of your MATLAB Production Server installation allows you to give any number of deployable archives their own exclusive pools of workers.

You select the deployable archives with an ECMAScript regular expression that matches the stem of the corresponding filename (for a file named `myctf.ctf`, the stem is `myctf`). For these isolated deployable archives, no other archive, whether isolated or non-isolated, will ever use this pool of workers. The server restarts workers in these isolated pools during a hot-deploy update of an existing deployable archive.

This property can be used to enable certain functionality.

- Prevent a Deployable Archive from Consuming All Available Workers
  - If you want to separate longer-running archives to prevent them from consuming all available workers, you can use the `component-isolation` property to assign them their own pools of workers.
  - *Example:*

    Assuming deployable archives named `long_running_1, long_running_2, short_running_1, short_running_2`

    ```
    --component-isolation component-match=^long_running_;max-workers=3
    --component-isolation component-match=^short_running_;max-workers=1
    --component-isolation max-isolated-components=4
    ```
- Separate Low-Priority and High-Priority Deployable Archives
  - If you want to prioritize the execution of certain components, you can use the `component-isolation` property to assign them their own pools of workers with a higher `max-workers` value than your lower-priority archives.

    *Example:*

    Assuming components named `high_priority_1, high_priority_2, med_priority_1, med_priority_2, low_priority_3, low_priority_3`

    ```
    --component-isolation component-match=^high_priority_;max-workers=3
    --component-isolation component-match=^med_priority_;max-workers=2
    --component-isolation component-match=^low_priority_;max-workers=1
    --component-isolation max-isolated-components=6
    ```

You can also use the `component-isolation` feature to avoid certain issues.

*Examples:*

- Third-party software, such a MEX file, that is not same-process safe (e.g. python code loaded from 2 or more components in same process)
- Third-party software that is not hot-deploy safe. In other words, software that does not destruct or reinitialize correctly when a deployable archive is hot-deployed after at least one prior usage by the worker. One example of this that can happen if the software within the component CTF file contains static file or class scope data within native code and does not properly utilize the native code for these conditions. This could occur through improper use (or non-use) of the following software, among others reasons:

- Windows: `LoadLibrary`, `FreeLibrary`, `DllMain`
- Linux : `dlopen`, `dlcose`, `__attribute__((constructor))`

## See Also

## More About

- “Control Worker Restarts”
- “Support Multiple MATLAB Runtime Versions”