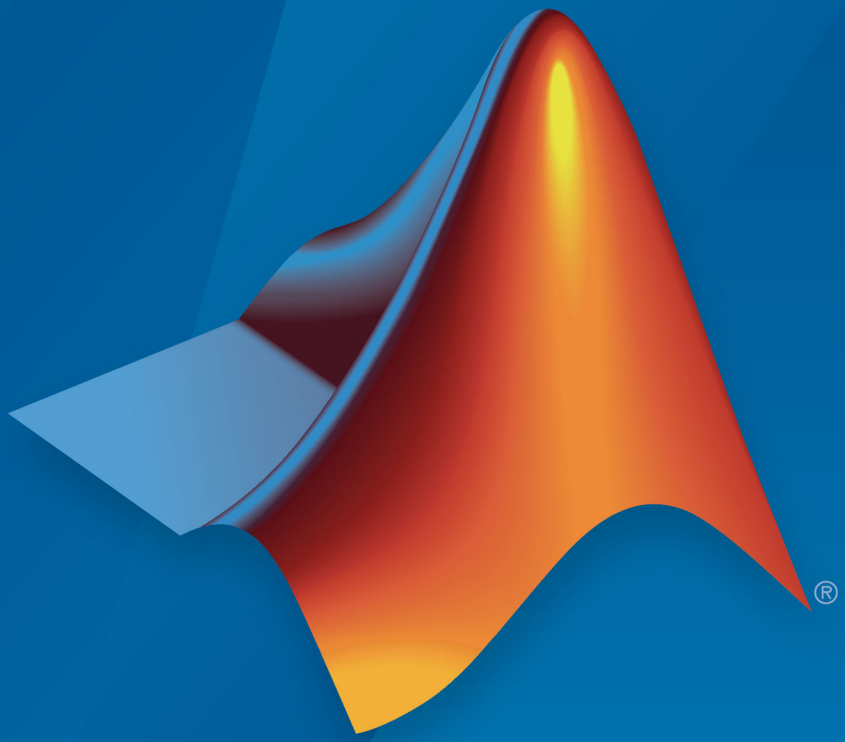
MATLAB® Production Server™

Getting Started



MATLAB®

R2017b

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.
3 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)

Contents

1 Overview

2 Installation

3 Set Up

4 Licensing

5 Deploying an Application

1

Overview

MATLAB Production Server Product Description

Integrate MATLAB analytics 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 a 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.

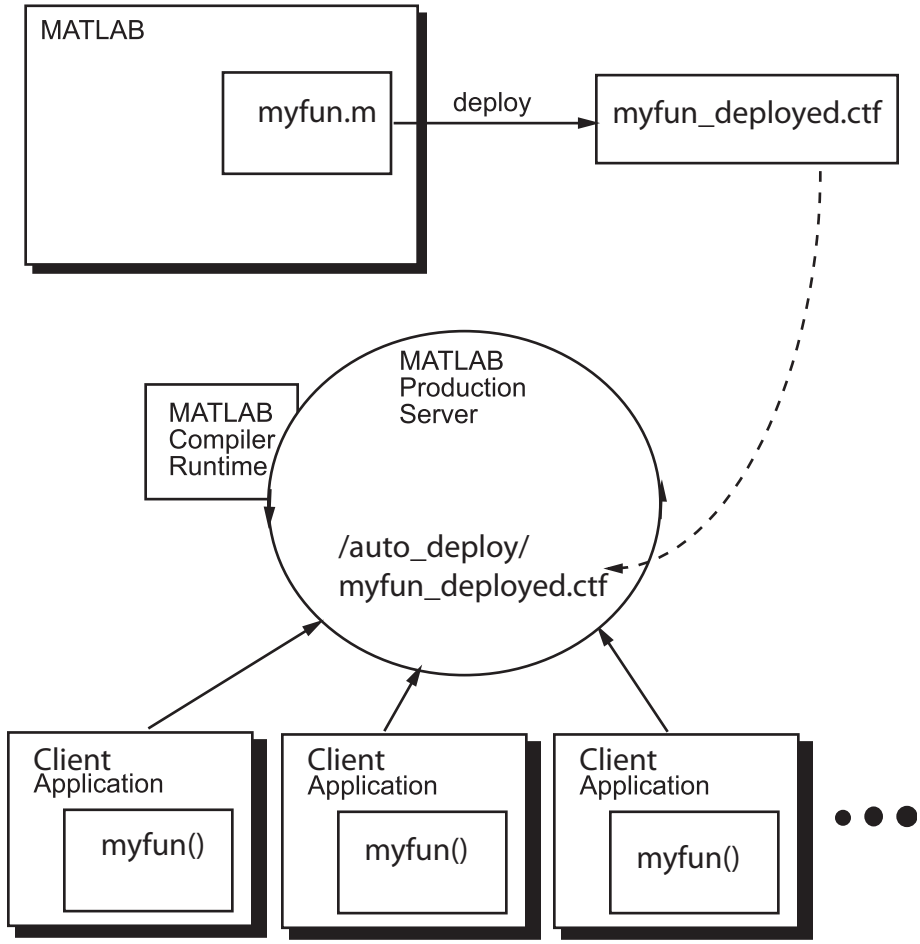
MATLAB Production Server manages multiple MATLAB Runtime versions simultaneously. As a result, algorithms developed in different versions of MATLAB can be incorporated into your application. The server runs on multiprocessor and multicore computers, providing low-latency processing of concurrent work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

Key Features

- Production deployment of MATLAB programs without recoding or creating custom infrastructure
- Scalable performance and management of MATLAB analytics and MATLAB Runtime versions
- Lightweight client library for secure access to analytics by enterprise applications
- Centralized analytic service accessible via the RESTful JSON interface or from .NET, Java®, C/C++, and Python® environments
- Web-based management dashboard for IT configuration and control

MATLAB Production Server Workflow

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



Deploying MATLAB code using MATLAB Production Server is a four-phase process:

1 Create deployable archives.

MATLAB users write MATLAB functions and compile them into deployable archives using MATLAB Compiler™.

2 Deploying the archives to an instance of the MATLAB Production Server.

   Server administrators take the deployable archives and deploy them into one or more instances of the MATLAB Production Server. In addition to adding the archive to a server's deployment folder, the server administrator might need to:

   - Install a server instance.
   - Set up licenses for a server instance.
   - Configure a server instance.
   - Install a MATLAB Runtime into a server instance.

3 Write client applications that use deployed MATLAB code via the server.

   Application developers use MATLAB Production Server client APIs to write applications that use MATLAB code.

4 Install client applications on end-user computers.

   Application installers distribute the client applications to the end-users.

Installation

Install MATLAB Production Server

1. Insert the installation DVD into your computer. If the MathWorks® Installer does not automatically start, run `setup.exe`.
2. Follow the instructions in the Installation Wizard. For help completing the wizard, see the *MATLAB Installation Guide*. As you run the installation wizard, note the following:

   - If you do not already have the License Manager installed, you must install it.
   - If you install the product using the internet, you will be taken to the Licensing Center to complete the licensing process.

Download and Install the MATLAB Runtime

The MATLAB Runtime is a standalone set of shared libraries that enables 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 it hosts.

Note Download and install the required version of the MATLAB Runtime from the Web at `http://www.mathworks.com/products/compiler/mcr`.

In order to host a deployable archive created with the Server Archive Compiler, you install a version of the MATLAB Runtime that is compatible with the version of MATLAB you used to create your archive.

For more information about the MATLAB compiler , including alternate methods of installing it, see “MATLAB Runtime” (MATLAB Compiler).

Disable Windows Interactive Error Reporting

If the system on which you are running MATLAB Production Server is not monitored frequently, you may want to disable Windows® Interactive Error Reporting, using the `DontShowUI` Windows Error Reporting (WER) setting, to avoid processing disruptions.

See WER Settings for Windows Development at `http://msdn.microsoft.com/en-us/library/windows/desktop/bb513638(v=vs.85).aspx` for complete information.

Set Up

Create a Server

In this section...

Prerequisites

Before creating a server, ensure you have:

- Installed MATLAB Production Server software 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.

> **Note** You can run server commands from the `script` folder. The script folder is located at *`$MPS_INSTALL`*`\script`, where *`$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

Before you can deploy your MATLAB code with MATLAB Production Server, you need to create a server to host your deployable archive.

A server instance is considered to be 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.

To create a server configuration or instance, do the following:

1. From the system command prompt, navigate to where you want to create your server instance.
2. Enter the `mps-new` command from the system prompt:

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

   where:

- *`path`* is the path to the server instance and configuration you want to create for use with the MATLAB Production Server product. When specifying a path, ensure the path ends with the *`server_name`*.

  If you are creating a server instance in the current folder, you do not need to specify a full path. Only specify the server name.
- *`server_name`* — is the name of the server instance and configuration you want to create.
- `-v` — enables verbose output, giving you information and status about each folder created in the server configuration.

Upon successful completion of the command, MATLAB Production Server creates a new server instance.

Specify the Default MATLAB Runtime for New Server Instances

Each server that you create with MATLAB Production Server has its own configuration file that defines various server management criteria.

The `mps-setup` command line wizard searches for MATLAB Runtime instances and sets the default path to the MATLAB Runtime for all server instances you create with the product.

To run the command line wizard, do the following after first downloading and performing the "Download and Install the MATLAB Runtime" on page 2-3:

1. Ensure you have logged on with `administrator` privileges.
2. At the system command prompt, run `mps-setup` from the `script` folder.

   Alternately, 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 is `C:\Program Files\MATLAB\MATLAB Production Server\`*`ver`*`\script\mps-setup`.

   `ver` is the version of MATLAB Production Server to use.
3. Follow the instructions in the command line wizard.

   The wizard will search your system and display installed MATLAB Runtime instances.
4. Enter `y` to confirm or `n` to specify a default MATLAB Runtime for all server configurations created with MATLAB Production Server.

   If `mps-setup` cannot locate an installed MATLAB Runtime on your system, you will be prompted 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 run `mps-setup`, specify the path name of the MATLAB Runtime as a command line argument. For example, on Windows:

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

mcrver is the version of the MATLAB Runtime to use.

Start a Server Instance

In this section...

Prerequisites

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

- Installed the MATLAB Runtime on page 2-3
- Created a server instance on page 3-2
- Specified the default MATLAB Runtime for the instance on page 3-4

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 you want to create. *`path`* should end with the server name.
- *`server_name`* — Name of the server instance you want to start or stop.
- `-f` — Forces command to succeed, regardless or whether the server is already started or stopped.

Upon successful completion of the command, the server instance is active.

Note If needed, use the `mps-status` command to verify the server is running.

Verify Server Status

In this section...

Procedure

To verify the status of a server instance, complete the following steps:

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 and configuration you want to create. *`path`* should end with the server name.
- *`server_name`* — Name of the server instance and configuration you want to start or stop.

Upon successful completion of the command, the server status displays.

License Server Status Information

In addition to the status of the server, `mps-status` also displays the status of the license server associated with the server you are verifying.

Possible statuses and their meanings follow:

This License Server Status Message...	Means...
`License checked out`	The server is operating with a valid license. The server is communicating with the License Manager, and the proper number of license keys are checked out.

This License Server Status Message...	Means...
`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 connectivity to the license server has not been restored, request processing will be disabled 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. Request processing has been suspended, but the server actively attempts to reestablish communication with the License Manager until it succeeds, at which time normal request processing resumes.
`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. Request processing has been suspended, but the server actively attempts to reestablish communication with the License Manager until it succeeds, at which time normal request processing resumes.

Verify Status of a Server

This example shows how to verify the status of the server instance you started in the previous example.

In this example, you verify the status of `prod_server_1`, from a location other than the server instance folder (`C:\tmp\prod_server_1`).

1 Open a system command prompt.

2 To verify `prod_server_1` is running, enter this command:

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

If `prod_server_1` is running, the following status is displayed:

```
\tmp\prod_server_1 STARTED
license checked out
```

This output confirms `prod_server_1` is running and the server is operating with a valid license.

For more information on the `STOPPED` status, see `mps-stop` and `mps-restart`.

For more information about license status messages, see “License Server Status Information” on page 3-7.

Licensing

Manage Licenses for MATLAB Production Server

Complete instructions for installing License Manager can be found in the *MATLAB Installation Guide*.

In addition to following instructions in the License Center to obtain and activate your license, do the following in order to set up and manage licensing for MATLAB Production Server:

Specify or Verify License Server Options in Server Configuration File

Specify or verify values for License Server options in the server configuration file (`main_config`). You create a server by using the `mps-new` command.

Edit the configuration file for the server. Open the file *`server_name`*`/config/main_config` and specify or verify parameter values for the following options. See the comments in the server configuration file for complete instructions and default values.

- `license` — Configuration option to specify the license servers and/or the 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 specify this value for `license`:

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

  The system searches these resources in this 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 MATLAB Production Server responds to HTTP requests, after license server heartbeat has been lost. See FLEXlm® 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 license server is polled, to verify and checkout a valid license. Polling occurs at the interval specified by `license-poll-interval` until license has been successfully checked-out. See FLEXlm documentation for more on heartbeats and related license terminology.

Verify Status of License Server using mps-status

When you enter an `mps-status` command, the status of the server *and* the associated license is returned.

For detailed descriptions of these status messages, see "License Server Status Information" on page 3-7.

Forcing a License Checkout Using mps-license-reset

Use the `mps-license-reset` command to force MATLAB Production Server to checkout a license. You can use this command at any time, providing 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`.

Deploying an Application

Create a Deployable Archive for MATLAB Production Server

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

To create a deployable archive:

1 Create a MATLAB function `addmatrix.m` that adds two matrices.

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

a = a1 + a2;
```

At the MATLAB command prompt, enter `addmatrix(1,2)`.

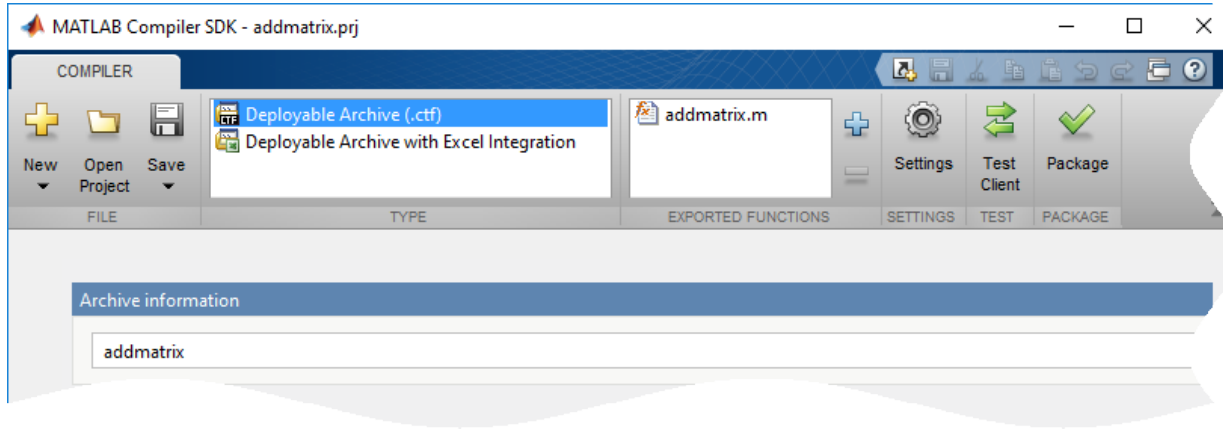
The output appears as follows:

```
ans =

     3
```

2 Open the **Production Server Compiler** app.

a On the toolstrip, select the **Apps** tab.
b Click the arrow on the far right of the tab to open the apps gallery.
c Click **Production Server Compiler**.



3 In the **Application Type** section of the toolstrip, select **Deployable Archive** from the list.

Note If the **Application Type** section of the toolstrip is collapsed, you can expand it by clicking the down arrow .

4 Specify the MATLAB functions you want to deploy.

a In the **Exported Functions** section of the toolstrip, click the plus button.

Note If the **Exported Functions** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

b Using the file explorer, locate and select the `addmatrix.m` file.

c Click **Open** to select the file and close the file explorer.

5 Explore the main body of the project window.

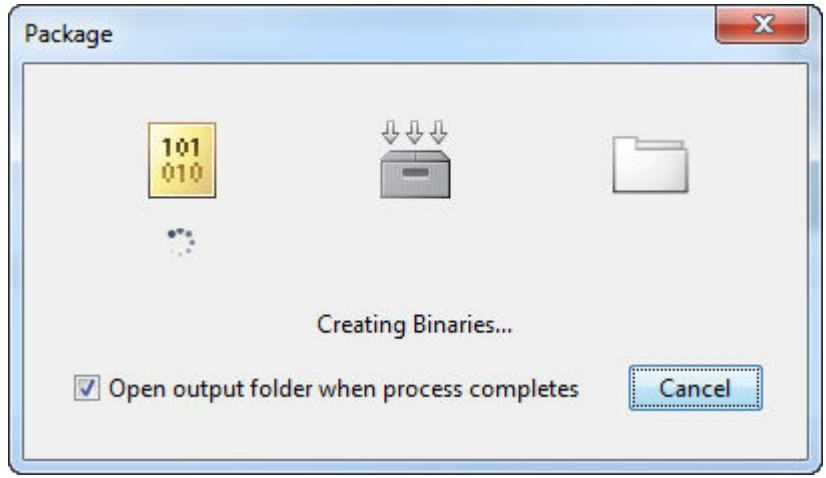
The project window is divided into the following areas:

- **Archive Information** — Editable information about the deployed archive.
- **Files required for your archive to run** — Additional files required by the archive. These files will be included in the generated archive. See “Manage Required Files in Compiler Project”.
- **Files packaged with the archive** — Files that are packaged with your archive. These files include:
  - `readme.txt`
  - `addmatrix.ctf` file

  See “Specify Files to Install with Application”.

6 Click **Package**.

The Package window opens while the library is being generated.



7 Select the **Open output folder when process completes** check box.

When the deployment process is complete, a file explorer opens and displays the generated output.

8 Verify the contents of the generated output:

- `for_redistribution` — A folder containing the installer to redistribute the archive to the system administrator responsible for the MATLAB Production Server
- `for_testing` — A folder containing the raw files generated by the compiler
- `PackagingLog.txt` — A log file generated by the compiler.

9 Click **Close** on the Package window.

To lean more about MATLAB Production Server see “MATLAB Production Server”

Start a MATLAB Production Server Instance

In this section...

Overview

This example shows how to install, configure, and start an instance of MATLAB Production Server.

To start a MATLAB Production Server instance:

1. Install MATLAB Production Server on page 5-5
2. Install MATLAB Runtime on page 5-6
3. "Create a Server Instance" on page 5-6
4. Configure the server instance on page 5-6
5. Start the server instance on page 5-7

Install MATLAB Production Server

To install MATLAB Production Server:

1. Run the installer.
2. Select License Manager for installation in the product list.
3. When asked where to install MATLAB Production Server, enter the name of an empty folder.

   You need the path to the installation to complete the tutorial.
4. Add the *`$MPS_INSTALL`*`\script` folder to your system `PATH` environment variable.

   `$MPS_INSTALL` represents your MATLAB Production Server installation folder.

Install MATLAB Runtime

If it is not already installed on your system, you must install the MATLAB Runtime. MATLAB Production Server requires the MATLAB Runtime.

To install a MATLAB Runtime:

1. Download the MATLAB Runtime installer from http://www.mathworks.com/products/compiler/mcr.
2. Run the MATLAB Runtime installer.

Create a Server Instance

To create the server instance:

1. Move to the folder where you want to create your server.
2. Run the `mps-new` command.

   ```
   C:\tmp>mps-new prod_server_1 -v
   ```
3. Verify the output.

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

For more information on these folders, see “Server Diagnostic Tools”.

Configure the Server Instance

After you create a new server instance, you must configure it. The MATLAB Production Server configuration file, `main_config`, includes many parameters you can use to tune server performance. At a minimum, you must use the file to specify the location of the MATLAB Runtime you want to use with the server instance

To configure the server instance’s default MATLAB Runtime:

1 From the system command line, run `mps-setup`.
2 Follow the directions to specify which MATLAB Runtime the server instances uses.

For more information about configuration options, see “Edit the Configuration File”.

Start the Server

To start the server:

1 Run the `mps-start` command.

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

2 Verify the server instance has started using the `mps-status` command.

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

```
'C:\tmp\prod_server_1' STARTED
 license checked out
```

Share a Deployable Archive on the Server Instance

To make your deployable archive available using MATLAB Production Server, you must copy the deployable archive into 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.

To share the deployable archive created in “Create a Deployable Archive for MATLAB Production Server” on page 5-2, copy the deployable archive from the deployment project’s `for_redistribution` folder into the server’s `auto_deploy` folder.

Create a Java Client

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”.

- 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 *`$MPS_INSTALL`*`/client` folder, where *`$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 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 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();
        }
    }
}
```

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 "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:

```
java -classpath .;"MPS_INSTALL_ROOT\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
```

Create a C# Client

This example shows how to call a deployed MATLAB function from a C# application using MATLAB Production Server.

In your C# code, you must:

- Create a Microsoft® Visual Studio® Project.
- Create a Reference to the Client Run-Time Library.
- Design the .NET interface in C#.
- Write, build, and run the C# application.

This task is typically performed by .NET application programmer. This part of the tutorial assumes you have Microsoft Visual Studio and .NET installed on your computer.

Create a Microsoft Visual Studio Project

1. Open Microsoft Visual Studio.
2. Click **File > New > Project**.
3. In the New Project dialog, select the project type and template you want to use. For example, if you want to create a C# Console Application, select **Windows** in the **Visual C#** branch of the **Project Type** pane, and select the **C# Console Application** template from the **Templates** pane.
4. Type the name of the project in the **Name** field (`Magic`, for example).
5. Click **OK**. Your `Magic` source shell is created, typically named `Program.cs`, by default.

Create a Reference to the Client Run-Time Library

Create a reference in your `MainApp` code to the MATLAB Production Server client run-time library. In Microsoft Visual Studio, perform the following steps:

1. In the Solution Explorer pane within Microsoft Visual Studio (usually on the right side), select the name of your project, `Magic`, highlighting it.
2. Right-click `Magic` and select **Add Reference**.
3. In the Add Reference dialog box, select the **Browse** tab. Browse to the MATLAB Production Server client runtime, installed at *`$MPS_INSTALL`*`\client\dotnet`. Select `MathWorks.MATLAB.ProductionServer.Client.dll`.

4 Click **OK**. `MathWorks.MATLAB.ProductionServer.Client.dll` is now referenced by your Microsoft Visual Studio project.

Design the .NET Interface in C#

In this example, you invoke `mymagic.m`, hosted by the server, from a .NET client, through a .NET interface.

To match the MATLAB function `mymagic.m`, design an interface named `Magic`.

For example, the interface for the `mymagic` function:

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

might look like this:

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

Note the following:

- The .NET interface has the same number of inputs and outputs as the MATLAB function.
- You are deploying one MATLAB function, therefore you define one corresponding .NET method in your C# code.
- Both MATLAB function and .NET interface process the same types: input type `int` and the output type two-dimensional `double`.
- You specify the name of your deployable archive (`magic`, which resides in your `auto_deploy` folder) in your URL, when you call `CreateProxy` (`"http://localhost:9910/magic"`).

Write, Build, and Run the .NET Application

Create a C# interface named `Magic` in Microsoft Visual Studio by doing the following:

1 Open the Microsoft Visual Studio project, `MagicSquare`, that you created earlier.

2 In `Program.cs` tab, paste in the code below.

Note 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`)

```
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();
       }
     }
   }
}
```

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
```

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 "Create a Deployable Archive for MATLAB Production Server" on page 5-2 and deployed in "Share a Deployable Archive on the Server Instance" on page 5-8.

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/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 `$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/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
```

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 "Create a Deployable Archive for MATLAB Production Server" on page 5-2 and deployed in "Share a Deployable Archive on the Server Instance" on page 5-8.

Create a Python MATLAB Production Server client application:

1. Copy the contents of the *`MPS_INSTALL`*`\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()
```