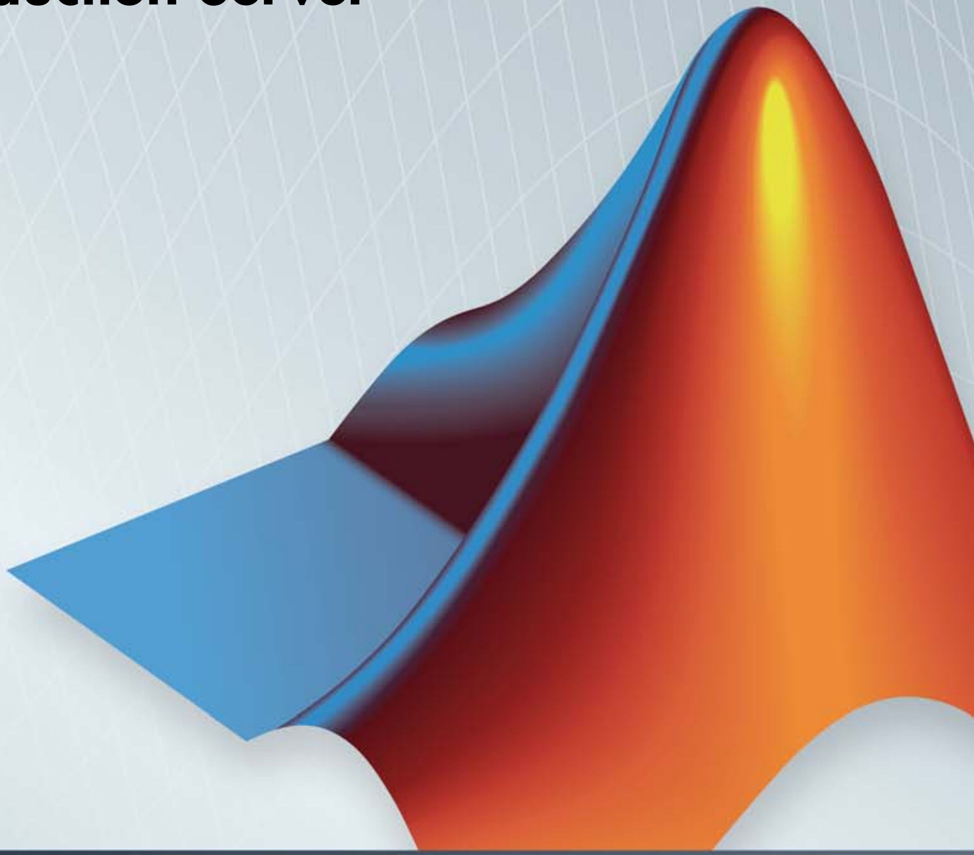
MATLAB® Production Server™

User's Guide

R2013**b**



MATLAB®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB® Production Server™

Trademarks

Patents

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)
September 2013	Online only	Revised for Version 1.0.2 (Release R2013b)

Contents

1 Getting Started with MATLAB Production Server

2 Write Deployable MATLAB Code

3 Create a Deployable CTF Archive from MATLAB Code

4 Customizing a Compiler Project

5 Server Management

6 Client Programming

7 Java Client Programming

8 .NET Client Programming

9 Commands — Alphabetical List

A Data Conversion Rules

B MATLAB Production Server .NET Client API Classes and Methods

Getting Started with MATLAB Production Server

MATLAB Production Server Product Description

Run MATLAB® programs as part of web, database, and enterprise applications

MATLAB Production Server™ lets you run MATLAB programs within your production systems, enabling you to incorporate numerical analytics in enterprise applications. Web, database, and enterprise applications connect to MATLAB programs running on MATLAB Production Server via a lightweight client library, isolating the MATLAB programs from your production system. You use MATLAB Compiler™ to package programs and deploy them directly to MATLAB Production Server without recoding or creating custom infrastructure to manage them. MATLAB Production Server runs on multiprocessor and multicore servers, providing low-latency processing of many concurrent requests. You can deploy the product on additional computer servers to increase the number of concurrent requests the system can handle and to provide redundancy.

Key Features

- Production deployment of MATLAB programs without recoding or creating custom infrastructure
- Scalable performance and management of packaged MATLAB programs
- Lightweight client library for calling numerical processing programs from enterprise applications
- Common infrastructure across .NET and Java® development environments
- Isolation of MATLAB processes from other system elements

Roles in Deploying to MATLAB Production Server

Deploying MATLAB functionality using MATLAB Production Server is a multistep process that might involve one or more team members. Each step requires fulfilling specific roles, as shown in MATLAB® Production Server™ Deployment Roles on page 1-3 .

MATLAB Production Server Deployment Roles

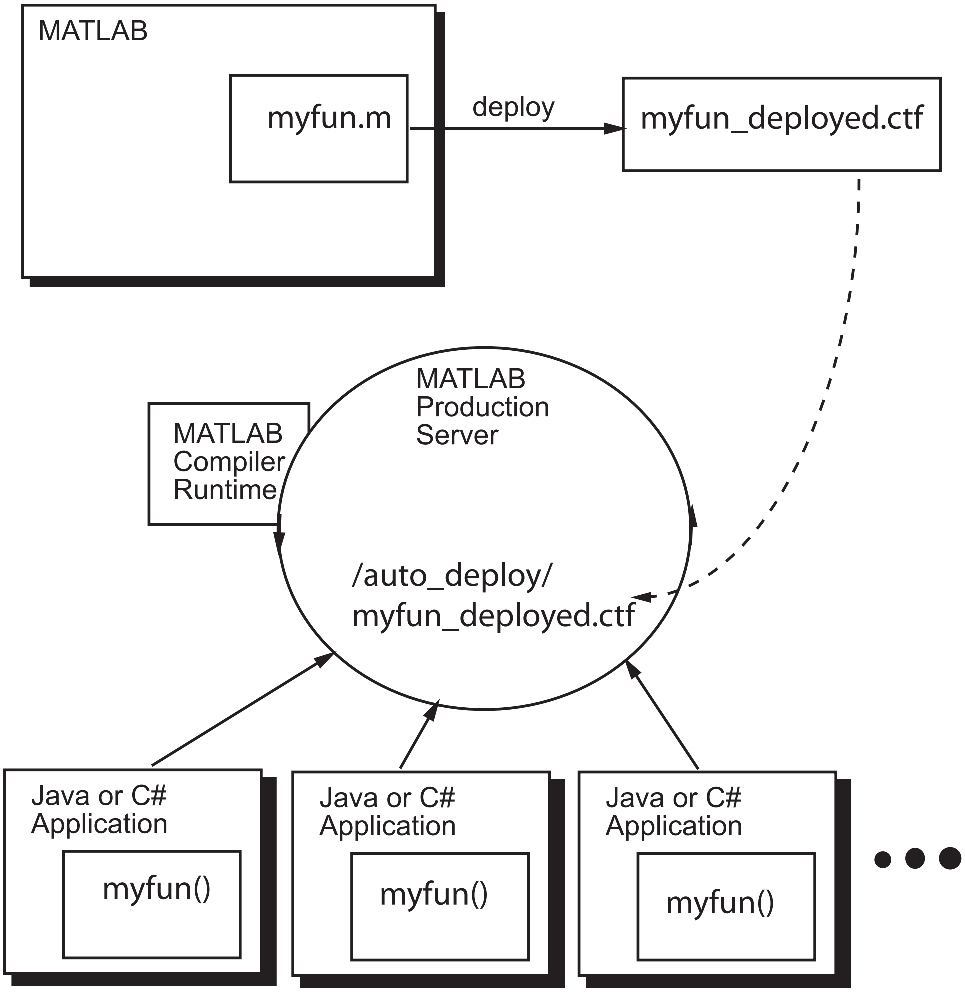
Role	Knowledge Base	Responsibilities
MATLAB programmer	• MATLAB expert • Little to no software development experience • Little to no IT experience	• Develop functions and implements them in MATLAB. • Create deployable archives (CTF Archive) that are deployed into MATLAB Production Server.
Application developer	• Little to no MATLAB experience • Some knowledge of IT systems • Familiarity with developing applications using a client/server architecture	• Develop applications using one of the MATLAB Production Server client APIs. • Test applications. • Package applications for distribution.

MATLAB Production Server Deployment Roles (Continued)

Role	Knowledge Base	Responsibilities
Server administrator	• Little to no MATLAB experience • Moderate IT experience • Familiarity with IT systems	• Ensure that systems running MATLAB Production Server instances have the required specifications. • Install MATLAB Production Server instances. • Tune MATLAB Production Server instances. • Install compiled MATLAB applications into MATLAB Production Server instances. • Monitor MATLAB Production Server instances.
Application installer	• Little to no MATLAB experience • Moderate IT experience • Familiarity with IT systems	• Ensure that systems using MATLAB Production Server client applications meet the required specifications. • Install any required software on target machines. • Install MATLAB Production Server client applications on target machines.

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 CTF 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 CTF 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 an MCR 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. MATLAB Production Server client APIs are available for:

- Java
- C#
- REST

4 Install client applications on end-user computers.

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

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 CTF archive:

1 In MATLAB, examine the MATLAB code that you want to deploy.

   a Open `addmatrix.m`.

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

   a = a1 + a2;
   ```

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

   The output appears as follows:

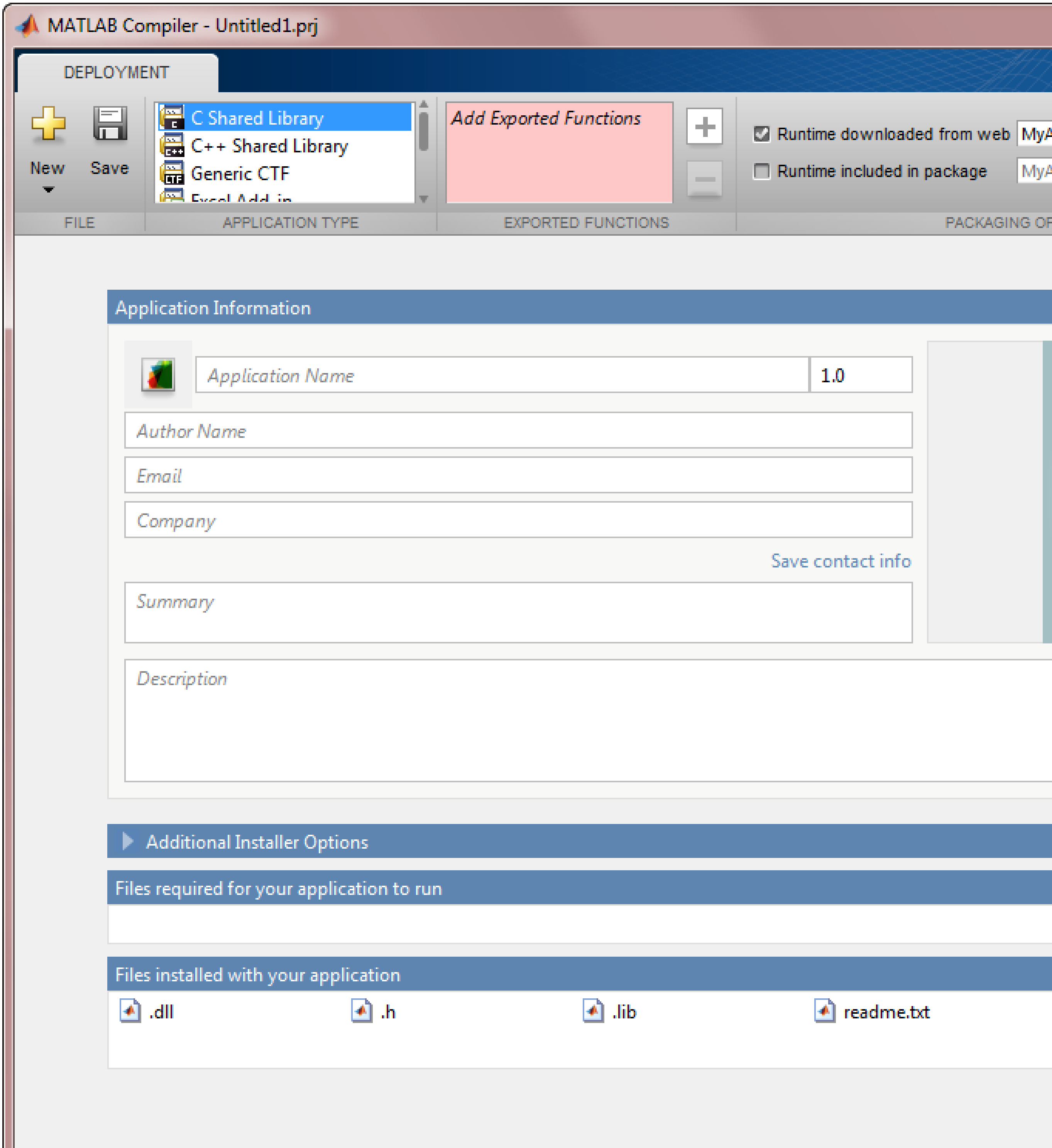
   ```
   ans =

        3
   ```

2 Open the **Library Compiler**.

   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 **Library Compiler** to open the **MATLAB Compiler** project window.



MATLAB Compiler - Untitled1.prj

C Shared Library

☑ Runtime downloaded from web MyA

☐ Runtime included in package MyA

1.0

3 In the **Application Type** section of the toolstrip, select **Generic CTF** 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.

   `addmatrix.m` is located in *`matlabroot`*`\extern\examples\compiler`.

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

   addmatrix.m is added to the field. A minus button will appear below the plus button.

5 In the **Packaging Options** section of the toolstrip, verify that the **Runtime downloaded from web** check box is selected.

Note If the **Packaging Options** section of the toolstrip is collapsed you can expand it by clicking the down arrow.

This option creates an application installer that automatically downloads the MATLAB Compiler Runtime (MCR) and installs it.

6 Explore the main body of the **MATLAB Compiler** project window.

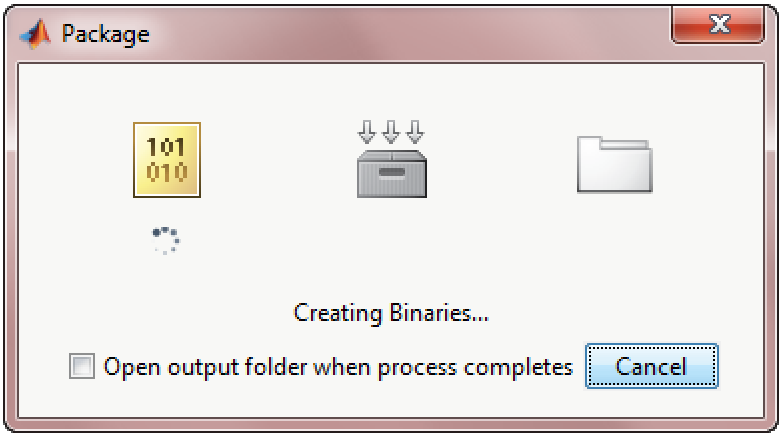
The project window is divided into the following areas:

- **Application Information** — Editable information about the deployed archive. This information is used by the generated installer to populate the installed application's metadata. See "Customizing the Installer" on page 4-2.
- **Additional Installer Options** — The default installation path for the generated installer. See "Customizing the Installer" on page 4-2.
- **Files required for your application** — Additional files required by the archive. These files will be included in the generated archive. See "Manage the Required Files Compiled into a Project" on page 4-6.
- **Files installed with your application** — Files that are installed with your archive. These files include:
  - `readme.txt`
  - `.ctf` file

  See "Specify Additional Files to Be Installed with the Application" on page 4-8.

7 Click **Package**.

The Package window opens while the library is being generated.



Creating Binaries...

8 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.

9 Verify the contents of the generated output:

- `for_redistribution` — A folder containing the installer to distribute the archive to the system administrator responsible for the MATLAB Production Server
- `for_testing` — A folder containing the raw files generated by the compiler
- `for_redistribution_files_only` — A folder containing only the files needed to redistribute the archive

10 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.
2. Install MATLAB Compiler Runtime (MCR).
3. Create a server instance.
4. Configure the server instance.
5. Start the server instance.

Install MATLAB Production Server

To install MATLAB Production Server:

1. Run the installer.
2. On the Installation Type dialog box, select **Custom**.
3. Select License Manager for installation in the product list.

4. 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.

5. Add the *`$MPS_INSTALL`*`\script` folder to your system `PATH` environment variable.

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

   For more information about the installation process, see .

Install MATLAB Compiler Runtime (MCR)

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

To install an MCR and configure MATLAB Production Server to use it:

1. Download the MCR installer from http://www.mathworks.com/products/compiler/mcr.

2. Run the MCR installer.

   For more information, see “Run mps-setup to Set Location of MATLAB Compiler Runtime (MCR)” on page 5-6.

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" on page 5-35.

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 MCR you want to use with the server.

To configure the server instance's default MCR:

1. From the system command line, run `mps-setup`.

2. Follow the directions to specify which MCR version MATLAB Production Server uses.

   For more information, see "Run mps-setup to Set Location of MATLAB Compiler Runtime (MCR)" on page 5-6.

   For more information about configuration options, see "Configuration File Customization" on page 5-22.

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 CTF Archive on the Server Instance

To make your CTF archive available using MATLAB Production Server, you must copy the deployable CTF 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 CTF archive created in "Create a Deployable Archive for MATLAB® Production Server™" on page 1-7, copy the deployable CTF archive from the deployment project's `for_redistribution_files_only` folder into the server's `auto_deploy` folder.

Create a Java Application That Calls the Deployed Function

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 `addmatrix_client.java`.
2. Using a text editor, open `addmatrix_client.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 .
- 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 CTF file and the name of your interface `class` as arguments:

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

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 (on one line):

```
H:\Work>javac -classpath "MPS_INSTALL_ROOT\client\java\mps_client.jar" addmatrix_client.java
```

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

For example, enter the following (on one line):

```
H:\Work>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# .NET Application That Calls a Deployed Function

This example shows how you can call a deployed MATLAB function from a C# application using MATLAB Production Server. For information about deploying a MATLAB function using Java, see .

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 generic CTF 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** Take care to ensure you reference the precise name of the CTF archive you are hosting on your server, as well as the port number where your server listens for client requests. For example, in the code below, the URL value (`"http://localhost:9910/mymagic_deployed"`) contains both CTF archive name (`mymagic_deployed`) and port number (`9910`).

C# Namespace Magic

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

namespace Magic
{
  public class MagicClass
  {

  class CustomConfig : MWHttpClientConfig
    {
     public int TimeoutMilliSeconds
        {
         get { return 120000; }
        }
  }
    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
```

Write Deployable MATLAB Code

Deployment Coding Guidelines

MATLAB coding guidelines are essentially the same for both the deployment products and MATLAB Production Server with important distinctions regarding functions that depend on MATLAB state.

Functions you deploy with MATLAB Production Server cannot be assumed to retain access to the same instance of the MATLAB Compiler Runtime, since the workers can access a number of different MCR instances. Therefore, when using MATLAB Production Server you must take extra care to ensure that state has not been changed or invalidated. See "State-Dependent Functions" on page 2-3 for more information.

Refer to "Write Deployable MATLAB Code" in the MATLAB Compiler documentation for general guidelines about deploying MATLAB code.

State-Dependent Functions

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

Does My MATLAB Function Carry State?

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

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

Defensive Coding Practices

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

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

The following are examples of "defensive coding" practices:

Reset System-Generated Values in the Deployed Application

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

Validate Global or Persistent Variable Values

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

Ensure Access to Data Caches

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

Use Simple Data Types When Possible

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

Avoid Using MATLAB Callback Functions

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

Techniques for Preserving State

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

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

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

Deploying MATLAB Functions Containing MEX Files

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

Coordinate with your server administrator and application developer as needed.

Unsupported MATLAB Data Types for Client and Server Marshaling

These data types are not supported for marshaling between MATLAB Production Server server instances and clients:

- MATLAB function handles
- Complex (imaginary) data
- Sparse arrays

Note See Appendix A, "Data Conversion Rules" for a complete list of conversion rules for supported MATLAB, .NET, and Java types.

Create a Deployable CTF Archive from MATLAB Code

Compile a Deployable CTF Archive with the Library Compiler App

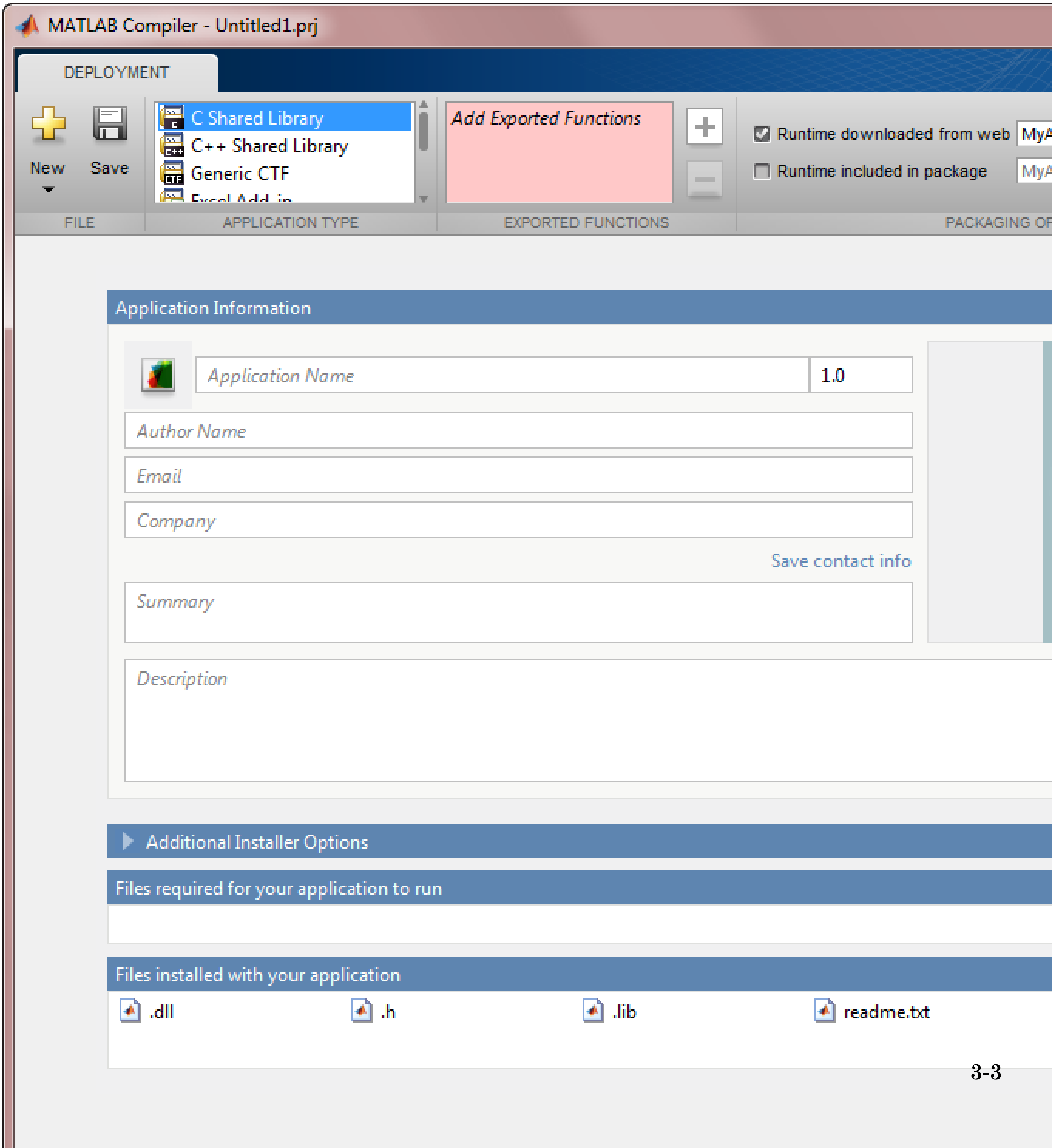
To compile MATLAB code into a deployable archive:

1 Open the **Library Compiler**.

   a On the toolstrip select the **Apps** tab on the toolstrip.

   b Click the arrow at the far right of the tab to open the apps gallery.

   c Click **Library Compiler** to open the **MATLAB Compiler** project window.



☑ Runtime downloaded from web

□ Runtime included in package

Note You can also launch the shared library compiler using the `libraryCompiler` function.

2 In the **Application Type** section of the toolstrip, select **Generic CTF**.

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

3 Specify the MATLAB files you want deployed in the package.

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 In the file explorer that opens, locate and select one or more the MATLAB files.

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

The names of the selected files are added to the list and a minus button appears below the plus button. The name of the first file listed is used as the default application name.

4 In the **Packaging Options** section of the toolstrip, specify how the installer will deliver the MATLAB Compiler Runtime (MCR) with the archive.

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

You can select one or both of the following options:

- **Runtime downloaded from web** — Generates an installer that downloads the MCR installer from the Web.
- **Runtime included in package** — Generates an installer that includes the MCR installer.

Note Selecting both options creates two installers.

Regardless of the options selected the generated installer scans the target system to determine if there is an existing installation of the appropriate MCR. If there is not, the installer installs the MCR.

5. Specify the name of any generated installers.

6. In the **Application Information** and **Additional Installer Options** sections of the compiler, customize the look and feel of the generated installer.

   You can change the information used to identify the application data used by the installer:

   - Splash screen
   - Application icon
   - Application version
   - Name and contact information of the archive's author
   - Brief summary of the archive's purpose
   - Detailed description of the archive

   You can also change the default location into which the archive is installed and provide some notes to the installer.

   All of the provided information is displayed as the installer runs.

   For more information see "Customizing the Installer" on page 4-2.

7 In the **Files required for your application to run** section of the compiler, verify that all of the files required by the deployed MATLAB functions are listed.

Note These files are compiled into the generated binaries along with the exported files.

In general the built-in dependency checker will automatically populate this section with the appropriate files. However, if needed you can manually add any files it missed.

For more information see "Manage the Required Files Compiled into a Project" on page 4-6.

8 In the **Files installed with your application** section of the compiler, verify that any additional non-MATLAB files you want installed with the application are listed.

Note These files are placed in the `applications` folder of the installation.

This section automatically lists:

- Generated deployable CTF archive
- Readme file

You can manually add files to the list. Additional files can include documentation, sample data files, and examples to accompany the application.

For more information see "Specify Additional Files to Be Installed with the Application" on page 4-8.

9 Click the **Settings** button to customize the flags passed to the compiler and the folders to which the generated files are written.

10 Click the **Package** button to compile the MATLAB code and generate any installers.

11 Verify that the generated output contains:

- `for_redistribution` — A folder containing the installer to distribute the archive
- `for_testing` — A folder containing the raw generated files to create the installer
- `for_redistribution_files_only` — A folder containing only the files needed to redistribute the archive

Compile a Deployable CTF Archive from the Command Line

In this section...

You can compile deployable archives from both the MATLAB command line and the system terminal command line:

- `deploytool` invokes the compiler app to execute a presaved compiler project
- `mcc` invokes the raw compiler

Execute Compiler Projects with deploytool

The `deploytool` command has two flags to invoke the compiler without opening a window:

- `-build` *`project_name`* — Invoke the compiler to build the project and do not generate an installer.
- `-package` *`project_name`* — Invoke the compiler to build the project and generate an installer.

For example, `deploytool -package magicsqaure` generates of the binary files defined by the `magicaqure` project and packages them into an installer that you can distribute to others.

Compile a Deployable CTF Archive with mcc

The `mcc` command invokes the raw compiler and provides fine-level control over the compilation of the deployable archive. It, however, cannot package the results in an installer.

To invoke the compiler to generate a deployable arcive use the `-W CTF:`*`component_name`* flag with `mcc`. The `-W CTF:`*`component_name`* flag creates a deployable archive called *`component_name`*`.ctf`.

For compiling deployable archives, you can also use the following options.

Compiler Java Options

Option	Description
`-a` *`filePath`*	Add any files on the path to the generated binary.
`-d` *`outFolder`*	Specify the folder into which the results of compilation are written.
`class{`*`className`*`:`*`mfilename`*`...}`	Specify that an additional class is generated that includes methods for the listed MATLAB files.

Modifying Deployed Functions

Once you have built a deployable CTF archive with the Deployment Tool, you can modify your MATLAB code, recompile with Deployment Tool, and see the change instantly reflected in the archive hosted on your server. This is known as "hot deploying" or "redeploying" a function.

To Hot Deploy, you must have a server created and running, with the built CTF archive located in the server's `auto_deploy` folder.

The server deploys the updated version of your archive when on the following occurs:

- Compiled archive has an updated time stamp
- Change has occurred to the archive contents (new file or deleted file)

It takes a maximum of five seconds to redeploy a function using Hot Deployment. It takes a maximum of ten seconds to undeploy a function (remove the function from being hosted).

To use Hot Deployment as default behavior for building deployable archives with the Deployment Tool, modify your Deployment Tool preferences to specify your `auto_deploy` folder as the output folder (`distrib` folder) location.

See for more information.

Customizing a Compiler Project

Customizing the Installer

In this section...

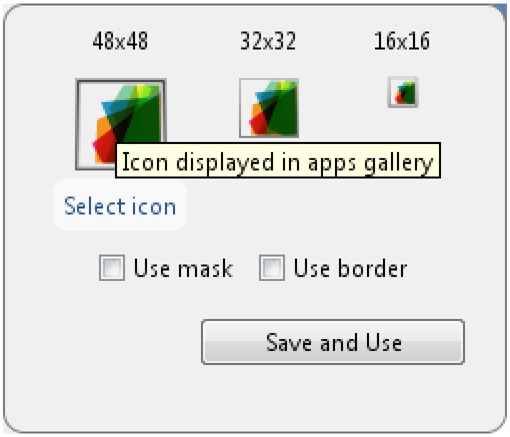
Changing the Application Icon

The application icon is used for the generated installer. For standalone applications, it is also the application's icon.

You can change the default icon in **Application Information**. To set a custom icon:

1 Click the graphic to the left of the **Application name** field.

A window previewing the icon opens.



2 Click **Select icon**.

3 Using the file explorer, locate the graphic file to use as the application icon.

4 Select the graphic file.

5 Click **OK** to return to the icon preview.

6 Select **Use mask** to fill any blank spaces around the icon with white.

7 Select **Use border** to add a border around the icon.

8 Click **Save and Use** to return to the main compiler window.

Adding Application Information

The **Application Information** section of the compiler app allows you to provide these values:

- Application name

  Determines the name of the installed executable file or shared library. For example, if the application name is `foo`, the installed executable would be `foo.exe`, the Windows® start menu entry would be **foo**. The folder created for the application would be *`InstallRoot`*`/foo`.

  The default value is the name of the first function listed in the **Main File(s)** field of the compiler.

- Application version

  The default value is 1.0.

- Author name
- Support e-mail address
- Company name

  Determines the full installation path for the installed executable or shared library. For example, if the company name is `bar`, the full installation path would be *`InstallRoot`*`/bar/`*`ApplicationName`*.

- Summary
- Description

This information is all optional and, unless otherwise stated, is only used for display purposes. It appears on the first page of the installer. On Windows systems, this information is also displayed in the Windows **Add/Remove Programs** control panel.

Changing the Splash Screen

The installer's splash screen displays after the installer is started. It is displayed, along with a status bar, while the installer initializes.

You can change the default image by clicking the **Select custom splash screen** link in **Application Information**. When the file explorer opens, locate and select a new image.

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

Changing the Installation Path

Default Installation Paths on page 4-4 lists the default path the installer will use when installing the compiled binaries onto a target system.

Default Installation Paths

Windows	`C:\Program Files\`*`companyName`*`\`*`appName`*
Mac OS X	`/Applications/`*`companyName`*`/`*`appName`*
Linux®	`/usr/`*`companyName`*`/`*`appName`*

You can change the default installation path by editing the **Default installation folder** field under **Additional Installer Options**.

The **Default installation folder** field has two parts:

- root folder — A drop down list that offers options for where the install folder is installed. Custom Installation Roots on page 4-5 lists the optional root folders for each platform.

Custom Installation Roots

Windows	`C:\Users\`*`userName`*`\AppData`
Linux	`/usr/local`

- install folder — A text field specifying the path appended to the root folder.

Changing the Application Logo

The application logo displays after the installer is started. It is displayed on the right side of the installer.

You change the default image by clicking the **Select custom logo** link in **Additional Installer Options**. When the file explorer opens, locate and select a new image.

Note You can drag and drop a custom image onto the default logo.

Editing the Installation Notes

Installation notes are displayed once the installer has successfully installed the packaged files on the target system. They can provide useful information concerning any additional set up that is required to use the installed binaries or simply provide instructions for how to run the application.

The field for editing the installation notes is in **Additional Installer Options**.

Manage the Required Files Compiled into a Project

In this section...

Dependency Analysis

The compiler uses a dependency analysis function to automatically determine what additional MATLAB files are required for the application to compile and run. These files are automatically compiled into the generated binary. The compiler does not generate any wrapper code allowing direct access to the functions defined by the required files.

Using the Compiler Apps

If you are using one of the compiler apps, the required files discovered by the dependency analysis function are listed in the **Files required by your application to run** field.

To add files:

1. Click the plus button in the field.
2. Select the desired file from the file explorer.
3. Click **OK**.

To remove files:

1. Select the desired file.
2. Press the **Delete** key.

Caution Removing files from the list of required files may cause your application to not compile or to not run properly when deployed.

Using mcc

If you are using `mcc` to compile your MATLAB code, the compiler does not display a list of required files before running. Instead, it compiles all of the required files that are discovered by the dependency analysis function and adds them to the generated binary file.

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

Specify Additional Files to Be Installed with the Application

The compiler apps packages additional files to be installed along with the ones it generates. By defaultthe installer includes a readme file with instructions on installing the MATLAB Compiler Runtine(MCR) and configuring it.

These files are listed in the **Files installed with your application** section of the app.

to add files to the list:

1. Click the plus (+) button in the field.
2. Select the desired file from the file explorer.
3. Click **OK** to close the file explorer.

To remove files from the list:

1. Select the desired file.
2. Press the **Delete** key.

Caution Removing the binary targets from the list results in an installer that does not install the intended functionality.

When installed on a target computer, the files listed in the **Files installed with your application** are placed in the `application` folder.

Server Management

Server Overview

In this section...

What is a Server?

You can create any number of server instances using MATLAB Production Server software. Each server instance can host any number of deployable archives containing MATLAB code. You may find it helpful to create one server for all archives relating to a particular application. You can also create one server to host code strictly for testing, and so on.

A *server instance* is considered to be one unique *configuration* of the MATLAB Production Server product. Each configuration has its own options file (`main_config`) and diagnostic files (`log` files, Process Identification (`pid`) files, and `endpoint` files).

In addition, each server has its own `auto_deploy` folder, which contains the deployable archives you want the server to host for clients.

The server also manages the MATLAB Compiler Runtime (MCR), which enables MATLAB code to execute. The settings in `main_config` determine how each server interacts with the MCR to process clients requests. You can set these parameters according to your performance requirements and other variables in your IT environment.

How Does a Server Manage Work?

A server processes a transaction using these steps:

1. The client sends MATLAB function calls to the master server process (the main process on the server).

2. MATLAB function calls are passed to one or more *MCR workers* (An MCR session).

3. MATLAB functions are executed by the MCR worker.

4. Results of MATLAB function execution are passed back to the master server process.

5. Results of MATLAB function execution are passed back for processing by the client.

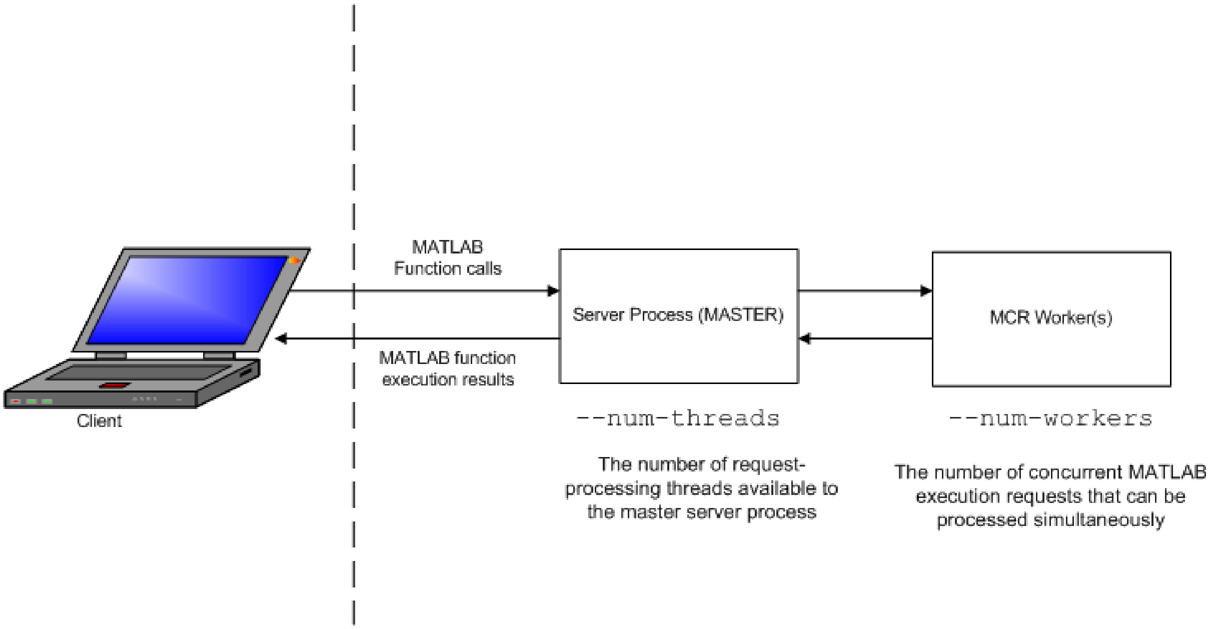
The server is the middleman in the MATLAB Production Server environment. It simultaneously accepts connections from clients, and then dispatches MCR workers—MATLAB sessions—to process client requests to the MCR. By defining and adjusting the number of workers and threads available to a server, you tune capacity and throughput respectively.

- Workers (capacity management) (`--num-workers`) — The number of MCR workers available to a server.

  Each MCR worker dispatches one MATLAB execution request to the MCR, interacting with one client at a time. By defining and tuning the number of workers available to a server, you set the number of concurrent MATLAB execution requests that can be processed simultaneously. `--num-workers` should roughly correspond to the number of cores available on the local host.

- Threads (throughput management) (`--num-threads`) — The number of threads (units of processing) available to the master server process.

  Throughput is the rate at which data moves during one complete pass from client to server (represented in figure MATLAB® Production Server™ Data Flow from Client to Server and Back on page 5-4).


MATLAB
Function calls
MATLAB function
execution results
Client
Server Process (MASTER)
MCR Worker(s)
--num-threads
--num-workers
The number of request-processing threads available to the master server process
The number of concurrent MATLAB execution requests that can be processed simultaneously

MATLAB® Production Server™ Data Flow from Client to Server and Back

The server does not allocate a unique thread to each client connection. Rather, when data is available on a connection, the required processing is scheduled on a *pool* of threads. `--num-threads` sets the size of that pool (the number of available request-processing threads) in the master server process. The threads in the pool do not execute MATLAB code directly. Instead, there is a single thread within each MCR worker process that executes MATLAB code on the client's behalf. The number of threads you define to a server should roughly correspond to the number of cores available on the local host.

Install MATLAB Production Server

Prerequisite for Windows Installations

If you plan to install on Windows, ensure that the system on which you install MATLAB Production Server does not depend on access to files located on a network drive. For stable results in a production environment, servers created with MATLAB Production Server should always have *local* access to the deployable CTF archives that they host.

Ensure Deployment Architecture Compatibility

Consider if the computers running MATLAB, as well as server instances of MATLAB Production Server that host your code, are 32-bit or 64-bit.

Your operating system and bit architectures must be compatible (or ideally, the same) across machines running MATLAB Production Server and your deployed components.

For additional compatibility considerations, see the MATLAB documentation.

Installing 32-Bit Version on 64-Bit Systems

You can install a 32-bit image of MATLAB Production Server on a 64-bit version of Windows.

If you do so, you will receive a message prompting you to run `set MPS_ARCH=win32`.

Run the Installation and Licensing Wizards

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. Select the **Custom** installation option to verify what will be installed. 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. On the Installation Type dialog box, choose to perform a **Custom** installation. Then, on the Product Selection dialog box, select the License Manager for installation. By default, it is not selected for installation.
- 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 Compiler Runtime (MCR)

The MATLAB Compiler Runtime (MCR) 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. When used together, MATLAB Production Server and the MCR enable you to create and distribute mathematical applications or software components quickly and securely.

Download and Install the latest version of the MATLAB Compiler Runtime (MCR) from the Web, on the MATLAB Compiler Runtime page at `http://www.mathworks.com/products/compiler/mcr`.

For more information about the MCR, including alternate methods of installing it, see “Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)” in the MATLAB Compiler section of MathWorks Documentation Center.

Compatibility Considerations for MATLAB Compiler Runtime (MCR) and Deployed Components

In order to deploy a generic CTF archive created with the Deployment Tool, you install a version of the MCR that is compatible with the version of MATLAB you used to create your archive.

Run mps-setup to Set Location of MATLAB Compiler Runtime (MCR)

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 MCR instances and sets the default path to the MATLAB Compiler Runtime (MCR) for all server instances you create with the product.

To run the command line wizard, do the following after first downloading and performing the "MATLAB Compiler Runtime (MCR) Installation" on page 5-14:

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 MCR instances.
4. Enter y to confirm or n to specify a default MCR for all server configurations created with MATLAB Production Server.

   If mps-setup cannot locate an installed MCR 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 MCR as a command line argument. For example, on Windows:

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

mcrver is the version of the MCR to use.

Disable Windows Interactive Error Reporting (Optional)

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.

License Management for MATLAB Production Server

Complete instructions for installing License Manager can be found in the *MATLAB Installation Guide*. See "License Manager Tasks" for information on how to start License Manager.

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`:

  ```
  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 5-31.

Forcing a License Checkout Using mps-license-reset

Use the `mps-license-reset` server 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`.

Server Creation

In this section...

Prerequisites

Before creating a server, verify that you have completed .

Procedure

Before you can deploy your MATLAB code with MATLAB Production Server software, you need to create a server instance 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 options file (`main_config`) and set of diagnostic files.

To create a server configuration or *instance*, complete the following steps:

1 From the system command prompt, navigate to where you want to create your server instance.

2 From the system prompt, enter the following command:

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

where:

- *`path`* — Path to the server instance and configuration you want to create

  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`* — 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 software creates a new server instance.

Create a Server

This example shows how to create a new server instance with the MATLAB Production Server software:

1 Select a folder where you want to create prod_server_1. For example, choose the /tmp folder, off your root.

```
cd /tmp
```

2 Enter the following command:

```
mps-new prod_server_1 -v
```

mps-new creates a new server instance named prod_server_1 in /tmp. Enabling the verbose (-v) option allows you to see the results of the command as each folder in the hierarchy is built.

The command produces the following output:

```
prod_server_1/.mps-version...ok
prod_server_1/config/...ok
prod_server_1/config/main_config...ok
prod_server_1/endpoint/...ok
prod_server_1/auto_deploy/...ok
prod_server_1/.mps-socket/...ok
prod_server_1/log/...ok
prod_server_1/pid/...ok
```

For more information on the files created by mps-new, see "Server Diagnostic Tools" on page 5-35.

Note Before using a server, you must start it. See "Server Startup" on page 5-27.

For More Information

For information about...	See...
How to solve errors when creating a server	"Server Troubleshooting" on page 5-34
The `mps-new` command	`mps-new` command reference page in the *MATLAB Production Server User's Guide*
Product installation	

MATLAB Compiler Runtime (MCR) Installation

If you already have the MATLAB Compiler Runtime (MCR) installed, skip this step and “Configuration File Customization” on page 5-22.

Install the MATLAB Compiler Runtime (MCR)

Download and Install the latest version of the MATLAB Compiler Runtime (MCR) from the Web, on the MATLAB Compiler Runtime page at `http://www.mathworks.com/products/compiler/mcr/`.

For more information about the MCR, including alternate methods of installing it, see “Distributing MATLAB Code Using the MATLAB Compiler Runtime (MCR)” in the MATLAB Compiler section of MathWorks Documentation Center.

Secure a Server

In this section...

Overview

MATLAB Production Server uses HTTPS to establish secure connections between server instances and clients. The HTTPS layer provides certificate-based authentication for both clients and server instances. It also provides an encrypted data path between the clients and server instances. The level of security provided by the HTTPS layer and the security protocols it supports are configurable.

MATLAB Production Server also provides a certificate-based authorization mechanism for restricting access to specific programs. Using this mechanism, you can specify the MATLAB programs a client can access.

Enabling Security

To enable security you need to add the following to the server instance's configuration:

- an HTTPS port
- a valid certificate stored in a PEM formatted certificate chain
- a valid private key stored in PEM format

The following configuration excerpt configures a server instance to accept secure connections on port 9920, use the certificate stored in

`./x509/my-cert.pem`, and use the unencrypted private key stored in `./x509/my-key.pem`.

```
...
--https 9920
--x509-cert-chain ./x509/my-cert.pem
--x509-private-key ./x509/my-key.pem
...
```

The default security settings allow all clients to access all of the programs hosted by the server instance. The server instance does not authenticate the clients nor does it perform any authorization. The default settings enable all security protocols and enable all but the eNULL cipher suites.

Using an unencrypted private key is not recommended in production settings since it is inherently insecure. To use an encrypted private key, specify the passphrase for decrypting the private key in a file that is owner read only and use the `--x509-passphrase` property to tell the server instance about it.

```
...
--https 9920
--x509-cert-chain ./x509/my-cert.pem
--x509-private-key ./x509/my-key.pem
--x509-passphrase ./x509/my-passphrase
...
```

Configuring Client Authentication

To ensure that only trusted client applications have access to a server instance, configure the server to require client authentication. To do this:

1 Set the server instance's `--ssl-verify-peer-mode` configuration property to `verify-peer-require-peer-cert`.

2 Configure the server instance to use the system provided CA store, a server specific CA store, or both.

   The CA store used by the server instance are controlled by two configuration properties:

   - `--x509-ca-file-store` specifies a PEM formatted CA store to authenticate clients.

- `--x509-use-system-store` directs the server instance to use the system's CA store to authenticate clients.

  Warning **`--x509-use-system-store` does not work on Windows.**

3. Optionally configure the server instance to respect any certificate revocation lists(CRLs) in the CA store.

   This behavior is specified by adding the `--x509-use-crl` property to the server's configuration. If this property is not specified, the server instance will ignore the CRLs and potentially authenticate clients using expired, or revoked, credentials.

   Warning **You must add a CRL list to the server's CA store before adding the `-x509-use-crl` property. If the the CA store does not include a CRL list, the server will crash.**

   The following configuration excerpt configures a server instance to authenticate clients using the system CA store and to respect CRLs:

   ```
   ...
   --https 9920
   --x509-cert-chain ./x509/my-cert.pem
   --x509-private-key ./x509/my-key.pem
   --x509-passphrase ./x509/my-passphrase
   --ssl-verify-peer-mode verify-peer-require-cert
   --x509-use-system-store
   --x509-use-crl
   ...
   ```

Specifying Access to MATLAB Programs

By default, server instances allow all clients to access all hosted MATLAB programs. You control this behavior using the server instance's `--ssl-allowed-client` configuration property. The `--ssl-allowed-client` property specifies a comma separated list of client's, identified its certificate's common name, that are allowed to access MATLAB programs. You also use the property to list specific MATLAB programs that a client is allowed to access.

If the --ssl-allowed-client property is not specified, the server instance does not restrict access to the hosted MATLAB programs. Once you add an entry for the --ssl-allowed-client property, the server instance only authorizes the listed clients to access the hosted MATLAB programs.

To only authorize client's with the common names jim, judy, and ash to use the MATLAB programs hosted on a server instance, you add the following configuration excerpt:

```
--ssl-allowed-client jim,judy,ash
```

You can restrict access further by only authorizing specific clients to have access to specific MATLAB programs. You do this by adding :*allowedPrograms* to the value of the --ssl-allowed-client property. *allowedPrograms* is a comma separated list of program names.

To allow clients with the common name jim access to all hosted programs, allow clients with the common name judy access to the programs tail and zap, and allow clients with the common name ash or joe access to the programs saw and travel, you add the following configuration excerpt:

```
--ssl-allowed-client jim
--ssl-allowed-client judy:tail,zap
--ssl-allowed-client ash,joe:saw,travel
```

Adjusting Security Protocols

By default, MATLAB Production Server instances will allow connections using SSLv2, SSLv3, and TLSv1. You can control the enabled protocols using the --ssl-protocols property. When you include this property in a server instance's configuration, it specifies the protocols the instance can use.

To enable only SSLv3, you add the following configuration excerpt:

```
--ssl-protocols SSLv3
```

Because SSLv2 and TLSv1 are not included in the list, the server instance does not enable the protocols.

You can similarly restrict the cipher suites used by the server instance using the --ssl-ciphers property. Once the property is added to a server instance's configuration, the server instance will only use the listed cipher suites.

To enable only high strength cipher suites, you add the following configuration excerpt:

```
--ssl-ciphers HIGH
```

Improving Start Up Time when Security is Activated

When a server instance is configured to use HTTPS, it generates an ephemeral DH key at start up. Generating the DH key at start up provides more security than reading it from a file on disk. However, this can add a couple of minutes to a server instance's start up time.

If you need the server instance to start up without delay and are not concerned about the loss of security, you can configure the server instance to read the ephemeral DH key from a file. This is done using the `--ssl-tmp-dh-param` configuration property. The `--ssl-tmp-dh-param` property specifies the file storing the DH key in PEM format.

Security Configuration Reference

Property	Description	Default
`--https`	Specifies the port number used for the secure connection.	
`--x509-private-key`	Specifies the path to the private key used to load the server's certificates. The private key **must** be in PEM format.	
`--x509-passphrase`	Specifies the path to the file containing the pass phase used to encrypt the private key. This file **must** be owner read only.	

Property	Description	Default
`--x509-cert-chain`	Specifies the path to the server's certificate chain file. This file contains the server's certificate and any untrusted certificates. This property requires that `--x509-private-key` also be set.	
`--ssl-verify-peer-mode`	Specifies if the server requires the client to provide a certificate for authentication. Valid values are: • `no-verify-peer` • `verify-peer-require-peer-cert`	`no-verify-peer`
`--x509-ca-file-store`	Specifies the path to the server's CA (certificate authority) store. This file contains trusted certificates that are used to verify client authenticity.	
`--x509-use-system-store`	Specifies that the server should use the host system's CA store. If this property is not set, the server will use the file specified by `--x509-ca-file-store`.	Do not use the system's CA store.
`--x509-use-crl`	Specifies if the server uses the certificate revocation list during client authentication.	Do not use the certificate revocation list.

Property	Description	Default
`--ssl-allowed-client`	Specifies a list of clients and the programs they can access. The format is `client1,client2, ...:program1,program2` where the clients in the list can access the programs in the list. If no program's are specified the listed clients can access all hosted programs.	Allow all clients to access all programs.
`--ssl-protocols`	Specifies a list of allowed protocols. Supported protocols include: • SSLv2 • SSLv3 • TLSv1	Allow all protocols.
`--ssl-ciphers`	Specifies a list of	`ALL`
`--ssl-tmp-dh-param`	Specifies that path to a pre-generated ephemeral DH key.	Server auto-generates an ephemeral DH key.

cipher suites except for the eNULL ciphers.

- `HIGH` — Enable all high encryption cipher suites.

Configuration File Customization

In this section...

Prerequisites

Before customizing the server configuration file with the location of the MATLAB Compiler Runtime (MCR), ensure you have:

- Created a server instance
- Installed the MATLAB Compiler Runtime

Procedure

Since the server interacts with the MCR to process client requests, you must specify where the MCR is located before you can start a server.

In addition, you set other critical configuration options that determine how a server hosts CTF archives and otherwise operates in a deployment environment.

You do this by editing `main_config`, the server configuration file for the server instance.

1. Navigate to the server instance you created. Open the top-most folder, labeled with the server name.

2. In the `config` folder, open `main_config` with a text editor of your choice.

3. In `main_config`, find the string `--mcr-root`, the configuration file option that designates the location of the MCR.

4. Specify the absolute path to the MCR, after entering one space after the option `--mcr-root`.

5. Save main_config and exit.

6. Run mps-restart to stop the server instance and start it with your specified options.

Use mps-setup to Locate Installed MCRs and Set --mcr-root

You can also use the mps-setup command to locate installed instances of the MATLAB Compiler Runtime (MCR) and specify the location of the MCR to the server instance.

For more information see the mps-setup command reference page.

Note mps-setup only sets --mcr-root. It does not set other options in main_config.

Specify the Installed MCR to the Server Instance

This example shows how to specify the installed location of the MATLAB Compiler Runtime (MCR) to your server instance. See "MATLAB Compiler Runtime (MCR) Installation" on page 5-14 for details about how to install the MCR.

1. Navigate to /tmp/prod_server_1.

2. Open the folder labeled prod_server_1.

3. Open the folder labeled config.

4. Open main_config with a text editor of your choice.

   Find the configuration file option --mcr-root in main_config. By default, in a new server instance, the value of --mcr-root is:

   ```
   mCRrOOTuNsET
   ```

5. Modify the --mcr-root option default value to point to the installed MCR you want to work with. For example:

   ```
   --mcr-root C:\Program Files\MATLAB\MATLAB Compiler Runtime\vnnn
   ```

Note You *must* specify the version number of the MCR (v*nnn*) in `--mcr-root`. MCR versions you specify must be compatible with MATLAB Production Server.

6 Save `main_config` and exit.

7 Run `mps-restart` to stop the server instance and start it with your specified options. To restart `prod_server_1` from a system command prompt, enter the following:

```
mps-restart -C /tmp/prod_server_1
```

About the Server Configuration File (main_config)

To change any MATLAB Production Server parameters, edit the `main_config` configuration file that corresponds to your specific server instance:

```
server_name/config/main_config
```

When editing `main_config`, remember these coding considerations:

- Each server has its own `main_config` configuration file.
- You enter only one configuration file parameter and related options per line. Each configuration file parameter starts with two dashes (`--`).
- Any line beginning with a pound sign (#) is ignored as a comment.
- Lines of white space are ignored.

Information about each configuration file parameter is included in the comments of each `main_config` file. The following are critical parameters to set or verify when running a server.

Setting the Location of the MATLAB Compiler Runtime (MCR). Use the `--mcr-root` parameter to specify the location of the MATLAB Compiler Runtime (MCR) to the server instance.

Setting Default Port Number for Client Requests. Use the `--http` parameter to set the default port number on which the server listens for client requests.

Setting Number of Available Workers. Use the `--num-workers` parameter to set the number of concurrent MATLAB execution requests that can be processed simultaneously.

See "Server Overview" on page 5-2 for more information.

Setting Number of Available Threads. Use the `--num-threads` parameter to set the number of request-processing threads available to the master server process.

See "Server Overview" on page 5-2 for more information.

Note For .NET Clients, the HTTP 1.1 protocol restricts the maximum number of concurrent connections from a client to a server to two.

This restriction only applies when the client and server are connected remotely. A local client and server connection has no such restriction.

To specify a higher number of connections than two for remote connection, use the NET classes `System.Net.ServicePoint` and `System.Net.ServicePointManager` to modify maximum concurrent connections.

For example, to specify four concurrent connections, code the following:

```
ServicePointManager.DefaultConnectionLimit = 4;
MWClient client = new MWHttpClient(new MyConfig());
MPSClient mpsExample = client.CreateProxy(new Uri("http://user01:9910/mpsex
```

For More Information

For information about...	See...
Downloading and installing the MCR	"MATLAB Compiler Runtime (MCR) Installation" on page 5-14
Product installation	

Server Startup

In this section...

Prerequisites

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

- Installed the MATLAB Compiler Runtime (MCR)
- Created a server
- Customized the server configuration file, `main_config` with the location of the MCR or Run mps-setup to set the location of the MATLAB Compiler Runtime (MCR)

Procedure

To start a server, complete the following steps:

1. Open a system command prompt.

2. Enter the following 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, query the status of the server instance that you started to verify the server is running.

Start a Server

This example shows how to start a server instance using the instance you created previously. In this example, you start `prod_server_1` from a location other than the server instance folder (`C:\tmp\prod_server_1`).

1 Open a system command prompt.

2 Enter the following command to start `prod_server_1`:

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

`prod_server_1` is now active and ready to receive requests.

For More Information

For information about...	See...
Downloading and installing the MCR	"Download and Install the MATLAB Compiler Runtime (MCR)" on page 5-6
Solving errors when starting or stopping a server	"Diagnose a Server Problem" on page 5-34 in the *MATLAB Production Server User's Guide.*
The `mps-start` command	`mps-start` command reference page in the *MATLAB Production Server User's Guide.*
Stopping the server with the `mps-stop` command	`mps-stop` command reference page in the *MATLAB Production Server User's Guide.*

For information about...	See...
Verifying status of a server with the `mps-status` command	`mps-status` command reference page in the *MATLAB Production Server User's Guide*.
Product installation	

Share the Deployable CTF Archive

After you create the deployable archive, share it with clients of MATLAB Production Server by copying it to your server, for hosting.

In order to share the deployable archive, a server must be created and started.

1. Locate your deployable archive in the `for_redistribution_files_only` folder of your Deployment Tool project folder. It will be named *`project_name`*`.ctf`.

2. Copy *`project_name`*`.ctf` to the `\`*`server_name`*`\auto_deploy` folder in your server instance.

   For example, if your server is named `prod_server_1` and located in `C:\tmp`, copy *`project_name`*`.ctf` to `C:\tmp\prod_server_1\auto_deploy`.

Note Once you deploy a MATLAB function using MATLAB Production Server, any future changes made to your MATLAB function, after recompiling with the Deployment Tool, are immediately available in the CTF archive that resides in the `auto_deploy` folder.

Server Status Verification

In this section...

Prerequisite

Before attempting to verify the status of a server instance, verify that you have first created a server.

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..
`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. For information about grace periods, see "Specify or Verify License Server Options in Server Configuration File" on page 5-9.

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 and the `mps-stop` command, see `mps-stop` and `mps-restart`

For more information about license status messages, see “License Server Status Information” on page 5-31.

For More Information

For information about....	See....
The `mps-status` command	`mps-status` command reference page in the *MATLAB Production Server User’s Guide*.
Stopping the server with the `mps-stop` command	`mps-stop` command reference page in the *MATLAB Production Server User’s Guide*.
Restarting the server with the `mps-restart` command	`mps-restart` command reference page in the *MATLAB Production Server User’s Guide*.

Server Troubleshooting

In this section...

Procedure

To diagnose a problem with a server instance or configuration of MATLAB Production Server, do the following, as needed:

- Check the logs for warnings, errors, or other informational messages.
- Check Process Identification Files (PID files) for information relating to problems with MCR worker processes.
- Check Endpoint Files for information relating to problems relating to the server's bound external interfaces — for example, a problem connecting a client to a server.
- Use server diagnostic tools, such as `mps-which`, as needed.

Diagnose a Server Problem

This example shows a typical diagnostic procedure you might follow to solve a problem starting server `prod_server_x`.

After you issue the command:

```
mps-start prod_server_x
```

from within the server instance folder (`prod_server_x`), you get the following error:

```
Server process exited with return code: 4
(check logs for more information)
Error while waiting for server to start: The I/O operation
```

```
has been aborted because of either a thread exit
or an application request
```

To solve this issue, you might check the `log` files for more detailed messages, as follows:

1. Navigate to the server instance folder (`prod_server_x`) and open the `log` folder.

2. Open `main.err` with any text editor. Note the following message listed under `Server startup error:`

   ```
   Dynamic exception type: class std::runtime_error
   std::exception::what: bad MCR installation:
   C:\Program Files\MATLAB\MATLAB Compiler Runtime\v717
   (C:\Program Files\MATLAB\MATLAB Compiler Runtime\v717\bin\
   win64\mps_worker_app could not be found)
   ```

3. The message indicates the installation of the MATLAB Compiler Runtime (MCR) is incomplete or has been corrupted. To solve this problem, reinstall the MCR.

Server Diagnostic Tools

Each server instance contains three sets of diagnostic files to help you determine and solve problems with the server and associated processes

Log Files

Each server writes a log file containing data from both the main server process, as well as the workers, named *`server_name`*`/log/main.log`. You can change the primary log folder name from the default value (`log`) by setting the option `--log-root` in `main_config`.

The primary log folder contains the `main.log` file, as well as a symbolic link to this file with the auto-generated name of `main_`*`date_fileID`*`.log`.

The `stdout` stream of the main server process is captured as `log/main.out`.

The `stderr` stream of the main server process is captured as `log/main.err`.

Log Retention and Archive Settings. Log data is written to the server's `main.log` file for as long as a specific server instance is active, or until midnight. When the server is restarted, log data is written to an archive log, located in the archive log folder specified by `--log-archive-root`.

You can set parameters that define when `main.log` is archived using the following options in each server's `main_config` file.

- `--log-rotation-size` — When `main.log` reaches this size, the active log is written to an archive log (located in the folder specified by `--log-archive-root`).
- `--log-archive-max-size` — When the combined size of all files in the archive folder (location defined by `--log-archive-root`) reaches this limit, archive logs are purged until the combined size of all files in the archive folder is less than `--log-archive-max-size`. Oldest archive logs are deleted first.

Specify values for these options using the following units and notations:

Represent these units of measure...	Using this notation...	Example
Byte	`b`	`900b`
Kilobyte (1024 bytes)	`k`	`700k`
Megabytes (1024 kilobytes)	`m`	`40m`
Gigabytes (1024 megabytes)	`g`	`10g`
Terabytes (1024 gigabytes)	`t`	`2t`
Petabytes (1024 terabytes)	`p`	`1p`

Note The minimum value you can specify for `--log-rotation-size` is 1 megabyte.

On Windows 32-bit systems, values larger than 2^{32} bytes are not supported. For example, specifying `5g` is not valid on Windows 32-bit systems.

Best Practices for Log Management. Use these recommendations as a guide when defining values for the options listed in “Log Retention and Archive Settings” on page 5-36.

- Avoid placing `--log-root` and `--log-archive-root` on different physical file systems.
- Place log files on local drives, not on network drives.
- Send MATLAB output to `stdout`. Develop an appropriate, consistent logging strategy following best MATLAB coding practices. See *MATLAB Programming Fundamentals* for guidelines.

Setting Log File Detail Levels. Each log level provides different levels of information for troubleshooting. For complete information on all logging levels and what details they provide, see the comments in the `main_config` file. Before you call support, you should set logging levels to `trace`.

Process Identification Files (PID Files)

Each process that the server runs generates a *Process Identification File (PID File)* in the folder identified as `pid-root` in `main_config`.

The main server PID file is `main.pid`; for each MCR Worker process, it is `worker-`*`n`*`.pid`, where *`n`* is the unique identifier of the worker.

PID files are automatically deleted when a process exits.

Endpoint Files

Endpoint files are generated to capture information about the server’s bound external interfaces. The files are created when you start a server instance and deleted when you stop it.

server_name/endpoint/http contains the IP address and port of the clients connecting to the server. This information can be useful in the event that zero (0) is specified in main_config, indicating that the server bind to a free port.

Common Error Messages and Resolutions

This section lists common troubleshooting scenarios, including error messages and typical resolutions:

(404) Not Found

Commonly caused by requesting a component that is not deployed on the server, or trying to call a function that is not exported by the given component.

Verify that the name of the CTF archive specified in your Uri is the same as the name of the CTF archive hosted in your auto_deploy folder.

Error: Bad MCR Instance

Common causes of this message include:

- You are not properly qualifying the path to the MCR. You must include the version number. For example, you need to specify:

  ```
  C:\Program Files\MATLAB\MATLAB Compiler Runtime\vn.n
  ```

  not

  ```
  C:\Program Files\MATLAB\MATLAB Compiler Runtime
  ```

Error: Server Instance not Specified

MATLAB Production Server can't find the server you are specifying.

Ensure you are either entering commands from the folder containing the server instance, or are using the -C command argument to specify a precise location of the server instance.

For example, if you created server_1 in C:\tmp\server_1, you would issue the mps-start command from within that folder to avoid specifying a path with the -C argument:

```
cd c:\tmp\server_1
mps-start server_1
```

For more information, see "Server Startup" on page 5-27.

Error: invalid target host or port

The port number specified has not been properly defined to your computer. Define a valid port and retry the command.

Error: HTTP error: HTTP/x.x 404 Component not found

This error can be caused by a number of reasons. Consult the "Log Files" on page 5-35 for further details on the precise cause of the problem.

For More Information

For information about....	See....
The `mps-status` command	`mps-status` command reference page in the *MATLAB Production Server User's Guide*
Displaying which server has allocated a client port with the `mps-which` command	`mps-which` command reference page in the *MATLAB Production Server User's Guide*

Client Programming

MATLAB Production Server Examples Available on MATLAB Central

Additional Client examples for MATLAB Production Server are available on MATLAB Central at `http://www.mathworks.com/matlabcentral`.

Create a MATLAB Production Server Client

To create a MATLAB Production Server client:

1. Obtain the client run-time files installed in in *$MPS_INSTALL*/client.
2. In consultation with the MATLAB programmer, agree on the MATLAB function signatures that comprise the services in the application.

   See the prerequisites section in and for specific requirements of each client.
3. Configure your system with the appropriate software for working with Java or .NET.
4. Write a the appropriate Java or .NET interface for the MATLAB functions the client uses.
5. Write a the Java or .NET code to instantiate a proxy to a MATLAB Production Server instance and call the MATLAB functions.
   - a Create a dynamic proxy for communicating with the service hosted by MATLAB Production Server software.
   - b Declare and throw exceptions as required.
   - c Free system resources using the `close` method of `MWClient`, after making needed calls to your application.

Unsupported MATLAB Data Types for Client and Server Marshaling

These data types are not supported for marshaling between MATLAB Production Server server instances and clients:

- MATLAB function handles
- Complex (imaginary) data
- Sparse arrays

Note See Appendix A, “Data Conversion Rules” for a complete list of conversion rules for supported MATLAB, .NET, and Java types.

Java Client Programming

Java Client Coding Best Practices

When you write Java interfaces to invoke MATLAB code, remember these considerations:

- The method name exposed by the interface *must* match the name of the MATLAB function being deployed.
- The method must have the same number of inputs and outputs as the MATLAB function.
- The method input and output types must be convertible to and from MATLAB.
- If you are working with MATLAB structures, remember that the field names are case sensitive and must match in both the MATLAB function and corresponding user-defined Java type.
- The name of the interface can be any valid Java name.
- Your code should support exception handling.

Java Client Prerequisites

Complete the following steps to prepare your MATLAB Production Server Java development environment.

1. Install a Java IDE of your choice. Follow instructions on the Oracle Web site for downloading Java, if needed.

2. Add `mps_client.jar` (located in *`$MPS_INSTALL`*`\client\java`) to your Java **`CLASSPATH`** and Build Path. This JAR file is sometimes defined in separate GUIs, depending on your IDE.

   Generate one generic CTF archive in your server's `auto_deploy` folder for each MATLAB application you plan to deploy. For information about creating a generic CTF archive with the Deployment Tool, see .

   Your server's `main_config` file should point to where MATLAB or your MCR instance is installed (using `mcr-root`).

3. The server hosting your deployable CTF archive must be running.

Manage Client Lifecycle

A single Java client connects to one or more servers available at various URLs. Even though you create multiple instances of MWHttpClient, one instance is capable of establishing connections with multiple servers.

Proxy objects communicate with the server until the close method of that instance is invoked.

For a locally scoped instance of MWHttpClient, the Java client code looks like the following:

Locally Scoped Instance

```
MWClient client = new MWHttpClient();
try{
    // Code that uses client to communicate with the server
}finally{
    client.close();
}
```

When using a locally scoped instance of MWHttpClient, tie it to a servlet.

When using a servlet, initialize the MWHttpClient inside the HttpServlet.init() method, and close it inside the HttpServlet.destroy() method, as in the following code:

Servlet Implementation

```
public class MPSServlet extends HttpServlet{
    private final MWClient client;

    public void init(ServletConfig config) throws ServletException{
        client = new MWHttpClient();
    }

    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException,java.io.IOException{

       // Code that uses client to communicate with the server
    }
    public void destroy(){
        client.close();
    }
}
```

Handling Java Client Exceptions

The Java interface must declare checked exceptions for the following errors:

Java Client Exceptions

Exception	Reason for Exception	Additional Information
`com.mathworks.mps.client.MATLABException`	A MATLAB error occurred when a proxy object method was executed.	The exception provides the following: • MATLAB Stack trace • Error ID • Error message
`java.io.IOException`	• A network-related failure has occurred. • The server returns an HTTP error of either `4xx` or `5xx`.	Use `java.io.IOException` to handle an HTTP error of `4xx` or `5xx` in a particular manner.

Managing System Resources

A single Java client connects to one or more servers available at different URLs. Instances of `MWHttpClient` can communicate with multiple servers.

All proxy objects, created by an instance of `MWHttpClient`, communicate with the server until the `close` method of `MWHttpClient` is invoked.

Call `close` only if you no longer need to communicate with the server and you are ready to release the system resources. Closing the client terminates connections to all created proxies.

Configure Client Timeout Value for Connection with a Server

To prevent client and server deadlocks and to ensure client stability, consider setting a timeout parameter when the client is connected with the server and the server becomes unresponsive.

To set a timeout parameter in milliseconds, implement interface MWHttpClientConfig in your client code. Use the overloaded constructor of MWHttpClient, which takes in an instance of MWHttpClientConfig.

Configure the following properties:

- **Interruptibility** — Determines if a MATLAB function call may interrupt while a client is waiting for a response from the server.
  - true — Allow interruptions
  - false — Do not allow interruptions
- **Timeout** — Time in milliseconds that the client is to wait for a response from the server before timing out.
- **Maximum connections per address** — The maximum amount of connections supported by one IP address. Default value is system-dependent—new connections are created for an address as long as it is supported by the system.

For example, to configure the client to:

- Allow interruptions from MATLAB function calls
- Timeout after no response from the server after 1.66 minutes (100000 milliseconds)
- Support a maximum of 10 connections per address

add the following to your client code.

```
MWClient client = new MWHttpClient(new MWHttpClientConfig(){
         public int getMaxConnectionsPerAddress(){
            return 10;
         }

         public long getTimeOutMs(){
            return 10000;
         }

         public boolean isInterruptible(){
            return true;
```

```
        }
});
```

Configuring Number of Reusable Connections

You can configure the number of reusable connections to the server in two ways:

- Use the default HTTP implementation (using the default construction of `MWHttpClient` without any input). Set the system property `http.maxConnections`. The value assigned to this property establishes the number of connections to reuse.
- Create `MWHttpClient` by passing an instance of `MWHttpClientConfig`. Set the Interuptibility property to `true` and set the number of maximum connections per address. This restricts the pool of open connections to the maximum value set.

Where to Find the Javadoc

The API doc for the Java client is installed in *`$MPS_INSTALL`*`/client`.

Bond Pricing Tool with GUI for Java Client

This example shows an application that calculates a bond price from a simple formula.

You run this example by entering the following known values into a simple graphical interface:

- Coupon payment — `C`
- Number of payments — `N`
- Interest rate — `i`
- Value of bond or option at maturity — `M`

The application calculates price (`P`) based on the following equation:

```
P = C * ( (1 - (1 + i)^-N) / i ) + M * (1 + i)^-N
```

Objectives

The Bond Pricing Tool demonstrates the following features of MATLAB Production Server:

- Deploying a simple MATLAB function with a fixed number of inputs and a single output
- Deploying a MATLAB function with a simple GUI front-end for data input
- Using `dispose()` to free system resources

Step 1: Write MATLAB Code

Implement the Bond Pricing Tool in MATLAB, by writing the following code. Name the code `pricecalc.m`.

Sample code is available in *`MPS_INSTALL`*`\client\java\examples\BondPricingTool\MATLAB`.

```
function price = pricecalc(value_at_maturity, coupon_payment,...
                           interest_rate, num_payments)

    C = coupon_payment;
    N = num_payments;
    i = interest_rate;
    M = value_at_maturity;

    price = C * ( (1 - (1 + i)^-N) / i ) + M * (1 + i)^-N;

end
```

Step 2: Create a Deployable CTF Archive with the Library Compiler App

To create the deployable archive for this example:

1. From MATLAB, select the Library Compiler App.
2. In the **Application Type** list, select **Generic CTF**.
3. In the **Exported Functions** field, add `pricecalc.m`.

   `pricecalc.m` is located in `*MPS_INSTALL*\client\java\examples\BondPricingTool\MATLAB`.
4. Under **Application Information**, change `pricecalc` to `BondTools`.
5. Click **Package**.

   The generated deployable archive, `BondTools.ctf` is located in the `for_redistribution_files_only` of the project's folder.

Step 3: Share the Deployable CTF Archive on a Server

1. Download the MATLAB Compiler Runtime, if needed, at http://www.mathworks.com/products/compiler/mcr. See "MATLAB Compiler Runtime (MCR) Installation" on page 5-14 for more information.
2. Create a server using `mps-new`. See "Server Creation" on page 5-11 for more information.
3. If you have not already done so, specify the location of the MATLAB Compiler Runtime (MCR) to the server by editing the server configuration

file, main_config and specifying a path for --mcr-root. See “Configuration File Customization” on page 5-22 for details.

4 Start the server using mps-start and verify it is running with mps-status.

5 Copy the BondTools.ctf file to the auto_deploy folder on the server for hosting. See for complete details.

Step 4: Create the Java Client Code

Create a compatible client interface and define methods in Java to match MATLAB function pricecalc.m, hosted by the server as BondTools.ctf, using the guidelines in this section.

Additional Java files are also included that are typical of a standalone application. You can find the example files in *MPS_INSTALL*\client\java\examples\BondPricingTool\Java.

This Java code...	Provides this functionality...
BondPricingTool.java	Runs the calculator application. The variable values of the pricing function are declared in this class.
BondTools.java	Defines pricecalc method interface, which is later used to connect to a server to invoke pricecalc.m
BondToolsFactory.java	Factory that creates new instances of BondTools
BondToolsStub.java	Java class that implements a dummy pricecalc Java method. Creating a stub method is a technique that allows for calculations and processing to be added to the application at a later time.
BondToolsStubFactory.java	Factory that returns new instances of BondToolsStub

This Java code...	Provides this functionality...
`RequestSpeedMeter.java`	Displays a GUI interface and accepts inputs using Java Swing classes
`ServerBondToolsFactory.java`	Factory that creates new instances of `MWHttpClient` and creates a proxy that provides an implementation of the `BondTools` interface and allows access to `pricecalc.m`, hosted by the server

When developing your Java code, note the following essential tasks, described in the sections that follow. For more information about clients coding basics and best practices, see .

This documentation references specific portions of the client code. You can find the complete Java client code in *`MPS_INSTALL`*`\client\java\examples\BondPricingTool\Java`.

Declare Java Method Signatures Compatible with MATLAB Functions You Deploy

To use the MATLAB functions you defined in "Step 1: Write MATLAB Code" on page 7-8, declare the corresponding Java method signature in the interface `BondTools.java`:

```
interface BondTools {
    double pricecalc (double faceValue,
                      double couponYield,
                      double interestRate,
                      double numPayments)
            throws IOException, MATLABException;
}
```

This interface creates an array of primitive `double` types, corresponding to the MATLAB primitive types (`Double`, in MATLAB, unless explicitly declared) in `pricecalc.m`. A one to one mapping exists between the input arguments in both the MATLAB function and the Java interface The interface specifies compatible type `double`. This compliance between the MATLAB and Java signatures demonstrates the guidelines listed in .

Instantiate MWClient, Create Proxy, and Specify Deployable CTF Archive

In the `ServerBondToolsFactory` class, perform a typical MATLAB Production Server client setup:

1 Instantiate `MWClient` with an instance of `MWHttpClient`:

```
...
    private final MWClient client = new MWHttpClient();
```

2 Call `createProxy` on the new client instance. Specify port number (`9910`) and the CTF archive name (`BondTools`) the server is hosting in the `auto_deploy` folder:

```
...
 public BondTools newInstance () throws Exception
    {
     return client.createProxy(new URL("http://user1.dhcp.mathworks.com:9910/BondTools"),
                                                                          BondTools.class);
    }
...
```

Use dispose() Consistently to Free System Resources

This application makes use of the Factory pattern to encapsulate creation of several types of objects.

Any time you create objects—and therefore allocate resources—ensure you free those resources using `dispose()`.

For example, note that in `ServerBondToolsFactory.java`, you dispose of the `MWHttpClient` instance you created in “Instantiate MWClient, Create Proxy, and Specify Deployable CTF Archive” on page 7-12 when it is no longer needed.

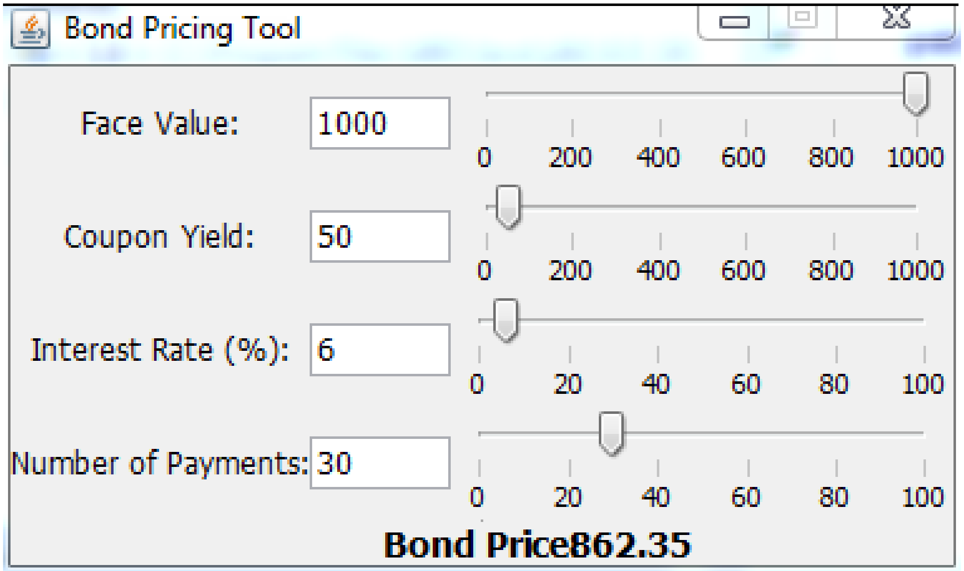
Additionally, note the `dispose()` calls to clean up the factories in `BondToolsStubFactory.java` and `BondTools.java`.

Step 5: Build the Client Code and Run the Example

Before you attempt to build and run your client code, ensure that you have done the following:

- Added `mps_client.jar` (*`$MPS_INSTALL`*`\client\java`) to your Java **`CLASSPATH`** and Build Path.
- Copied your deployable CTF archive to your server's `auto_deploy` folder.
- Modified your server's `main_config` file to point to where your MCR is installed (using `mcr-root`).
- Started your server and verified it is running.

When you run the calculator application, you should see the following output:



Face Value: 1000

Coupon Yield: 50

Interest Rate (%): 6

Number of Payments: 30

Bond Price862.35

Accessing Secure Programs Using HTTPS

In this section...

Overview

Connecting to a MATLAB Production Server instance over HTTPS provides a secure channel for executing MATLAB functions. To establish an HTTPS connection with a MATLAB Production Server instance:

1. Ensure that the server is configured to use HTTPS.
2. Install the required credentials on the client system.
3. Configure the client's Java environment to use the credentials.
4. Create the program proxy using the program's `https://` URL.

MATLAB Production Server's Java client API also provides:

- hooks for providing your own `HostnameVerifier` implementation
- hooks for implementing server authorization beyond that provided by HTTPS

Configuring the Client's Environment for SSL

Use `keytool` to manage the key store and trust stores on the client machine.

At a minimum the client requires the server's root CA (Certificate Authority) in its truststore.

If the client needs to connect to a server that requires client-side authentication, it also needs a signed certificate in its key store.

For information on using keytool see Oracle's keytool documentation.

Establishing an HTTPS Connection

You can create a secure proxy connection with a MATLAB Production Server instance by using the https:// URL for the desired program:

```
MWClient client = new MWHttpClient();
URL sslURL = new URL("https://hostname:port/myCTF");
MyCTFProxy sslProxy = client.createProxy(sslURL, MyCTFProxy.class);
```

The sslProxy object will use the default Java trust store, stored in JAVA_HOME\lib\security\cacerts, to perform the HTTPS server authentication. If the server requests client authentication, the HTTPS handshake will fail because the default SSLContext object created by the JRE does not provide a key store.

To enable your client to connect with a server instance requiring client authentication, you set the key store location and password using Java system properties:

```
System.setProperty("javax.net.ssl.keystore", "PATH_TO_KEYSTORE");
System.setProperty("javax.net.ssl.keystorePassword", "keystore_pass");
MWClient client = new MWHttpClient();
URL sslURL = new URL("https://hostname:port/myfun");
MyCTFProxy sslProxy = client.createProxy(sslURL,
MyCTFProxy.class);
```

To use a non-default location for the client trust store, you set the trust store location and password using Java system properties:

```
System.setProperty("javax.net.ssl.truststore", "PATH_TO_TRUSTSTORE");
System.setProperty("javax.net.ssl.truststorePassword", "truststore_pass");
MWClient client = new MWHttpClient();
URL sslURL = new URL("https://hostname:port/myfun");
MyCTFProxy sslProxy = client.createProxy(sslURL, MyCTFProxy.class);
```

To use a custom SSLContext implementation, add a custom HostnameVerifier implementation, or use the MATLAB Production Server Java API's server authorization, you must provide a custom implementation of the MWSSLConfig interface. See "Advanced Security Configuration" on page 7-16.

Advanced Security Configuration

SSL API Configuration

The Java API uses an `MWSSLConfig` object to get the information it needs to use HTTPS and perform the additional server authorization. The `MWSSLConfig` interface has three methods:

- `getSSLContext()` — Returns the `SSLContext` object
- `getHostnameVerifier()` — Returns the `HostnameVerifier` object to use if HTTPS hostname verification fails
- `getServerAuthorizer()` — Returns the `MWSSLServerAuthorizer` object to perform server authorization based on the server’s certificate

The Java API provides a default `MWSSLConfig` implementation, `MWSSLDefaultConfig`, which it uses when no SSL configuration is passed to the `MWHTTPClient` constructor. The `MWSSLDefaultConfig` object is implemented such that:

- `getSSLContext()` returns the default `SSLContext` object created by the JRE
- `getHostnameVerifier()` returns a `HostnameVerifier` implementation that always returns false. If the HTTPS hostname verification fails, this will not override the HTTPS layer’s decision.
- `getServerAuthorizer()` returns a `MWSSLServerAuthorizer` implementation that authorizes all MATLAB Production Server instances.

Override Default Hostname Verification

As part of the SSL handshake, the HTTPS layer attempts to match the hostname in the provided URL to the hostname provided in the server’s certificate. If the two hostnames do not match, the HTTPS layer calls the `HostnameVerifier.verify()` method as an additional check. The return value of the HostnameVerifier.verify() method determines if the hostname is verified.

The implementation of the HostnameVerifier.verify() method provided by the MWSSLDefaultConfig object always returns false. The result is that if the hostname in the URL and the hostname in the server certificate do not match, the HTTPS handshake fails.

To use a hostname verification scheme that is more robust, you can extend the MWSSLDefaultConfig class to return an implementation of HostnameVerifier.verify() that uses custom logic. For example, if you only wanted to generate one certificate for all of the servers on which MATLAB Production Server instances run, you could specify MPS for the certificate's hostname. Then your implementation of HostnameVerifier.verify() returns true if the certificate's hostname is MPS.

```
public class MySSLConfig extends MWSSLDefaultConfig {
  public HostnameVerifier getHostnameVerifier() {
    return new HostNameVerifier() {
      public boolean verify(String s, SSLSession sslSession) {
        if (sslSession.getPeerHost().equals("MPS"))
          return true;
        else
          return false;
      }
    }
  }
}
```

For more information on HostnameVerify see Oracle's Java Documentation.

Use Additional Server Authentication

After the HTTPS layer establishes a secure connection, a client can perform an additional authentication step before sending requests to a server. This additional authentication is performed by an implementation of the MWSSLServerAuthorizer interface. An MWSSLSServerAuthorizer implementation performs two checks to authorize a server:

- `isCertificateRequired()` determines if server's must present a certificate for authorization. If this returns true and the server has not provided a certificate, the client does not authorize the server.
- `authorize(Certificate serverCert)` uses the server's certificate to determine if the client authorizes the server to process requests.

The `MWSSLSServerAuthorizer` implementation returned by the `MWSSLDefaultConfig` object authorizes all servers without performing any checks.

To use server authentication extend the `MWSSLDefaultConfig` class and override the implementation of `getServerAuthorizer()` to return a `MWSSLSServerAuthorizer` implementation that does perform authorization checks.

Code Multiple Outputs for Java Client

MATLAB allows users to write functions that return multiple outputs.

For example, consider this MATLAB function signature:

```
function [out_double_array, out_char_array] =
               multipleOutputs (in1_double_array, in2_char_array)
```

In the MATLAB signature, `multipleOutputs` has two outputs (`out_double_array` and `out_char_array`) and two inputs (`in1_double_array` and a `in2_char_array`, respectively)—a double array and a char array.

In order to call this function from Java, the interface in the client program must specify the number of outputs of the function as part of the function signature.

The number of expected output parameters in defined as type integer (`int`) and is the first input parameter in the function.

In this case, the matching signature in Java is:

```
public Object[] multipleOutputs(int num_args, double[]
                     in1Double, String in2Char);
```

where `num_args` specifies number of output arguments returned by the function. All output parameters are returned inside an array of type `Object`.

Note When coding multiple outputs, if you pass an integer *as the first input argument* through a MATLAB function, you must wrap the integer in a `java.lang.Integer` object.

Note the following coding best practices illustrated by this example:

- Both the MATLAB function signature and the Java method signature using the name `multipleOutputs`. Both signatures define two inputs and two outputs.
- MATLAB Java interface supports direct conversion from Java double array to MATLAB double array and from Java string to MATLAB char array. For more information, see “Conversion of Java Types to MATLAB Types” on page A-2 and “Conversion of MATLAB Types to Java Types” on page A-4.

For more information, see .

Code Variable-Length Inputs and Outputs for Java Client

MATLAB supports functions with both variable number of input arguments (`varargin`) and variable number of output arguments (`varargout`).

MATLAB Production Server Java client supports the ability to work with variable-length inputs (`varargin`) and outputs (`varargout`). `varargin` supports one or more of any data type supported by MATLAB. See the *MATLAB Function Reference* for complete information on `varargin` and `varargout`.

For example, consider this MATLAB function:

```
function varargout = vararginout(double1, char2, varargin)
```

In this example, the first input is type double (`double1`) and the second input type is a char (`char2`). The third input is a variable-length array that can contain zero, or one or more input parameters of valid MATLAB data types.

The corresponding client method signature must include the same number of output arguments as the first input to the Java method.

Therefore, the Java method signature supported by MATLAB Production Server Java client, for the `varargout` MATLAB function, is as follows:

```
public Object[] vararginout(int nargout, double in1, String in2, Object... vararg);
```

In the `vararginout` method signature, you specify equivalent Java types for `in1` and `in2`.

The variable number of input parameters is specified in Java as `Object... vararg`.

The variable number of output parameters is specified in Java as return type `Object[]`.

Note the following coding best practices illustrated by this example:

- Both the MATLAB function signature and the Java method signature using the name `vararginout`. Both signatures define two inputs and two outputs.

- MATLAB Java interface supports direct conversion from Java double array to MATLAB double array and from Java string to MATLAB char array. For more information, see “Conversion of Java Types to MATLAB Types” on page A-2 and “Conversion of MATLAB Types to Java Types” on page A-4.

Marshal MATLAB Structures (Structs) in Java

Structures (or *structs*) are MATLAB arrays with elements accessed by textual field designators.

Structs consist of data containers, called *fields*. Each field stores an array of some MATLAB data type. Every field has a unique name.

A field in a structure can have a value compatible with any MATLAB data type, including a cell array or another structure.

In MATLAB, a structure is created as follows:

```
S.name = 'Ed Plum';
S.score = 83;
S.grade = 'B+'
```

This code creates a scalar structure (S) with three fields:

```
S =
    name: 'Ed Plum'
    score: 83
    grade: 'B+'
```

A multidimensional structure array can be created by inserting additional elements:

```
S(2).name = 'Toni Miller';
S(2).score = 91;
S(2).grade = 'A-';
```

In this case, a structure array of dimensions (1,2) is created. Structs with additional dimensions are also supported.

Since Java does not natively support MATLAB structures, marshaling structs between the server and client involves additional coding.

Marshaling a Struct Between Client and Server

MATLAB structures are ordered lists of name-value pairs. You represent them in Java with a class using fields consisting of the same case-sensitive names.

The Java class must also have public get and set methods defined for each field. Whether or not the class needs both get and set methods depends on whether it is being used as input or output, or both.

Following is a simple example of how a MATLAB structure can be marshaled between Java client and server.

In this example, MATLAB function sortstudents takes in an array of structures (see "Marshal MATLAB Structures (Structs) in Java" on page 7-23 for details).

Each element in the struct array represents different information about a student. sortstudents sorts the input array in ascending order by score of each student, as follows:

```
function sorted = sortstudents(unsorted)
% Receive a vector of students as input
% Get scores of all the students
scores = {unsorted.score};
% Convert the cell array containing scores into a numeric array or doubles
scores = cell2mat(scores);
% Sort the scores array
[s i] = sort(scores);
% Sort the students array based on the sorted scores array
sorted = unsorted(i);
```

Note Even though this example only uses the scores field of the input structure, you can also work with name and grade fields in a similar manner.

You compile sortstudents into a deployable CTF archive (scoresorter.ctf) using the Deployment Tool (see for details) and make it available on the server at http://localhost:9910/scoresorter for access by the Java Client (see).

Before defining the Java interface required by the client, define the MATLAB structure, `Student`, using a Java class.

`Student` declares the fields `name`, `score` and `grade` with appropriate types. It also contains `public get` and `set` functions to access these fields.

Java Class Student

```
public class Student{

    private String name;
    private int score;
    private String grade;

    public Student(){
    }

    public Student(String name, int score, String grade){
        this.name = name;
        this.score = score;
        this.grade = grade;
    }

    public String getName(){
        return name;
    }

    public void setName(String name){
        this.name = name;
    }

    public int getScore(){
        return score;
    }

    public void setScore(int score){
        this.score = score;
    }

    public String getGrade(){
        return grade;
    }

    public void setGrade(String grade){
        this.grade = grade;
    }

    public String toString(){
        return "Student:\n\tname : " + name +
               "\n\tscore : " + score + "\n\tgrade : " + grade;
    }
}
```

Note Note that this example uses the `toString` method for marshaling convenience. It is not required.

Next, define the Java interface `StudentSorter`, which calls method `sortstudents` and uses the `Student` class to marshal inputs and outputs.

Since you are working with a struct type, `Student` must be included in the annotation `MWStructureList` .

Java Interface StudentSorter

```
interface StudentSorter {
    @MWStructureList({Student.class})
    Student[] sortstudents(Student[] students)
            throws IOException, MATLABException;
}
```

Finally, you write the Java application (`MPSClientExample`) for the client:

1. Create `MWHttpClient` and associated proxy (using `createProxy`) as shown in .

2. Create an unsorted student struct array in Java that mimics the MATLAB struct in naming, number of inputs and outputs, and type validity in MATLAB. See for more information.

3 Sort the student array and display it.

Java ClientExample Class

```
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;
import com.mathworks.mps.client.annotations.MWStructureList;

interface StudentSorter {
    @MWStructureList({Student.class})
    Student[] sortstudents(Student[] students)
            throws IOException, MATLABException;
}

public class ClientExample {

    public static void main(String[] args){

        MWClient client = new MWHttpClient();
        try{
            StudentSorter s =
               client.createProxy(new URL("http://localhost:9910/scoresorter"),
                                                     StudentSorter.class );
            Student[] students = new Student[]{new Student("Toni Miller", 90, "A"),
                                               new Student("Ed Plum",     80, "B+"),
                                               new Student("Mark Jones",  85, "A-")};
            Student[] sorted = s.sortstudents(students);
            System.out.println("Student list sorted in the
                            ascending order of scores : ");
            for(Student st:sorted){
                System.out.println(st);
            }
        }catch(IOException ex){
            System.out.println(ex);
        }catch(MATLABException ex){
            System.out.println(ex);
        }finally{
            client.close();
        }
    }
}
```

Defining MATLAB Structures Only Used as Inputs

When defining Java structs as inputs, follow these guidelines:

- Ensure that the fields in the Java class match the field names in the MATLAB struct *exactly*. The field names are case sensitive.
- Use `public get` methods on the fields in the Java class. Whether or not the class needs both `get` and `set` methods for the fields depends on whether it is being used as input or output or both. In this example, note that when `student` is passed as an input to method `sortstudents`, only the `get` methods for its fields are used by the data marshaling algorithm.

As a result, if a Java class is defined for a MATLAB structure that is only used as an input value, the `set` methods are not required. This version of the `Student` class only represents input values:

Java Class Student with Struct as Input

```
public class Student{

    private String name;
    private int score;
    private String grade;

    public Student(String name, int score, String grade){
        this.name = name;
        this.score = score;
        this.grade = grade;
    }

    public String getName(){
        return name;
    }
    public int getScore(){
        return score;
    }

    public String getGrade(){
        return grade;
    }
}
```

Defining MATLAB Structures Only Used as an Output

When defining Java structs as outputs, follow these guidelines:

- Ensure that the fields in the Java class match the field names in the MATLAB struct *exactly*. The field names are case sensitive.
- Create a new instance of the Java class using the structure received from MATLAB. Do so by using `set` methods or `@ConstructorProperties` annotation provided by Java. `get` methods are not required for a Java class when defining output-only MATLAB structures.

An output-only `Student` class using `set` methods follows:

Java Class Student with Struct as Output

```
public class Student{

    private String name;
    private int score;
    private String grade;

    public void setName(String name){
        this.name = name;
    }

    public void setScore(int score){
        this.score = score;
    }

    public void setGrade(String grade){
        this.grade = grade;
    }
}
```

An output-only `Student` class using `@ConstructorProperties` follows:

Defining MATLAB structures for output using @ConstructorProperties annotation

```
public class Student{

    private String name;
    private int score;
    private String grade;

    @ConstructorProperties({"name","score","grade"})
    public Student(String n, int s, String g){
        this.name = n;
        this.score = s;
        this.grade = g;
    }
}
```

Note If both `set` methods and `@ConstructorProperties` annotation are provided, `set` methods take precedence over `@ConstructorProperties` annotation.

Defining MATLAB Structures Used as Both Inputs and Outputs

If the `Student` class is used as both an input and output, you need to provide `get` methods to perform marshaling to MATLAB. For marshaling from MATLAB, use `set` methods or `@ConstructorProperties` annotation.

Data Conversion with Java and MATLAB Types

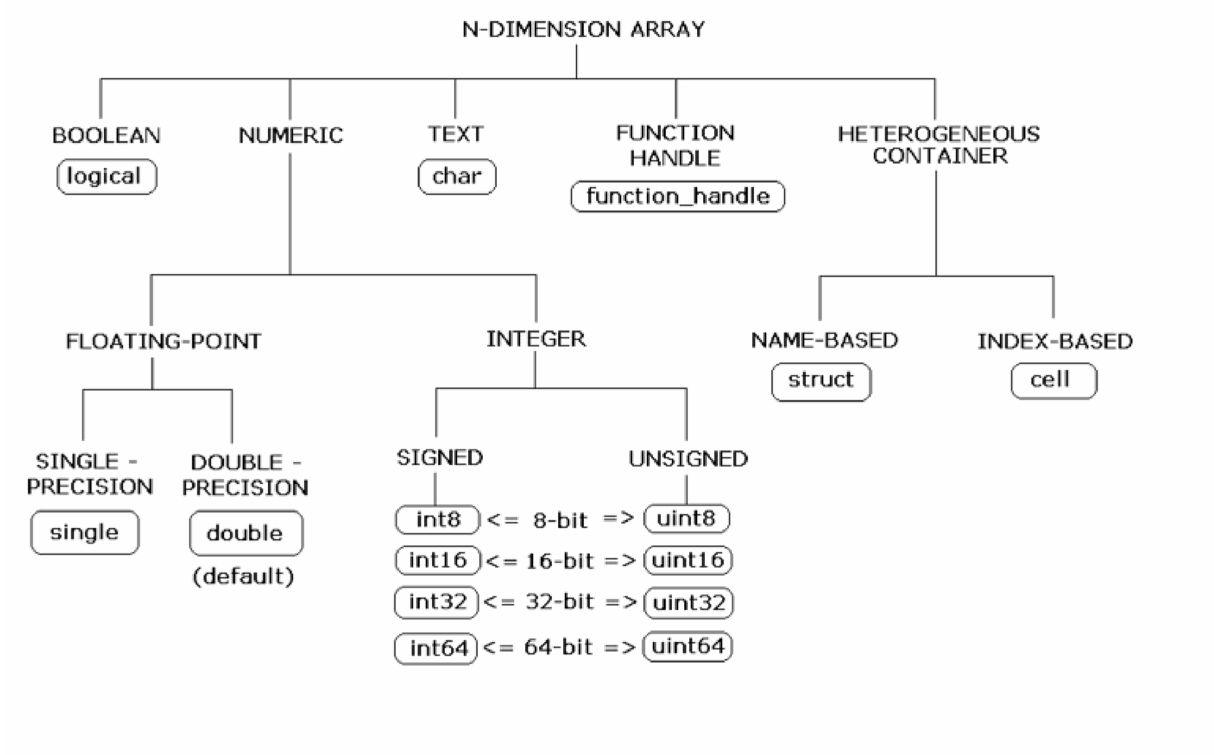
Working with MATLAB Data Types

There are many data types that you can work with in MATLAB. Each of these data types is in the form of a matrix or array. You can build matrices and arrays of floating-point and integer data, characters and strings, and logical true and false states. Structures and cell arrays provide a way to store dissimilar types of data in the same array.

All of the fundamental MATLAB classes are circled in the diagram Fundamental MATLAB Data Types on page 7-32.

The Java client follows *Java-MATLAB-Interface* (JMI) rules for data marshaling. It expands those rules for scalar Java boxed types, allowing auto-boxing and un-boxing, which JMI does not support.

Note Function Handles are not supported by MATLAB Production Server.



Fundamental MATLAB Data Types

The expected conversion results for Java to MATLAB types are listed in "Conversion of Java Types to MATLAB Types" on page A-2. The expected conversion results for MATLAB to Java types are listed in "Conversion of MATLAB Types to Java Types" on page A-4.

Scalar Numeric Type Coercion

Scalar numeric MATLAB types can be assigned to multiple Java numeric types as long as there is no loss of data or precision.

The main exception to this rule is that MATLAB `double` scalar data can be mapped into any Java numeric type. Because `double` is the default numeric type in MATLAB, this exception provides more flexibility to the users of MATLAB Production Server Java client API.

MATLAB to Java Numeric Type Compatibility on page 7-33 describes the type compatibility for scalar numeric coercion.

MATLAB to Java Numeric Type Compatibility

MATLAB Type	Java Types
`uint8`	`short`, `int`, `long`, `float`, `double`
`int8`	`short`, `int`, `long`, `float`, `double`
`uint16`	`int`, `long`, `float`, `double`
`int16`	`int`, `long`, `float`, `double`
`uint32`	`long`, `float`, `double`
`int32`	`long`, `float`, `double`
`uint64`	`float`, `double`
`int64`	`float`, `double`
`single`	`double`
`double`	`byte`, `short`, `int`, `long`, `float`

Dimensionality in Java and MATLAB Data Types

In MATLAB, dimensionality is an attribute of the fundamental types and does not add to the number of types as it does in Java.

In Java, `double`, `double[]` and `double[][][]` are three different data types. In MATLAB, there is only a `double` data type and possibly a scalar instance, a vector instance, or a multi-dimensional instance.

Java Signature	Value Returned from MATLAB
`double[][][] foo()`	`ones(1,2,3)`

Dimension Coercion

How you define your MATLAB function and corresponding Java method signature determines if your output data will be coerced, using padding or truncation.

This coercion is automatically performed for you. This section describes the rules followed for padding and truncation.

Padding

When a Java method's return type has more dimensions than MATLAB's, MATLAB's dimensions are be padded with ones (1s) to match the required number of output dimensions in Java.

You, as a developer, do not have to do anything to pad dimensions.

The following tables provide examples of how padding is performed for you:

How MATLAB Pads Your Java Method Return Type

When Dimensions in MATLAB are:	And Dimensions in Java are:	This Type in Java:	Returns this Type in MATLAB:
`size(a)` is`[2,3]`	Array will be returned as size `2,3,1,1`	`double [][][][] foo()`	`function a = foo a = ones(2,3);`

Padding Dimensions in MATLAB and Java Data Conversion

MATLAB Array Dimensions	Declared Output Java Type	Output Java Dimensions
`2 x 3`	`double[][][]`	`2 x 3 x 1`
`2 x 3`	`double[][][][]`	`2 x 3 x 1 x 1`

Truncation

When a Java method's return type has fewer dimensions than MATLAB's, MATLAB's dimensions are truncated to match the required number of output dimensions in Java. This is only possible when extra dimensions for MATLAB array have values of ones (1s) only.

To compute appropriate number of dimensions in Java, excess ones are truncated, in this order:

1. From the end of the array

2. From the array's beginning
3. From the middle of the array (scanning front-to-back).

You, as a developer, do not have to do anything to truncate dimensions.

The following tables provide examples of how truncation is performed for you:

How MATLAB Truncates Your Java Method Return Type

When Dimensions in MATLAB are:	And Dimensions in Java are:	This Type in Java:	Returns this Type in MATLAB
`size(a) is [1,2,1,1,3,1]`	Array will be returned as size `2,3`	`double [][] foo()`	`function a = foo a = ones(1,2,1,1,3,1);`

Following are some examples of dimension shortening using the `double numeric` type:

Truncating Dimensions in MATLAB and Java Data Conversion

MATLAB Array Dimensions	Declared Output Java Type	Output Java Dimensions
`1 x 1`	`double`	`0`
`2 x 1`	`double[]`	`2`
`1 x 2`	`double[]`	`2`
`2 x 3 x 1`	`double[][]`	`2 x 3`
`1 x 3 x 4`	`double[][]`	`3 x 4`
`1 x 3 x 4 x 1 x 1`	`double[][][]`	`1 x 3 x 4`
`1 x 3 x 1 x 1 x 2 x 1 x 4 x 1`	`double[][][][]`	`3 x 2 x 1 x 4`

Empty (Zero) Dimensions

Passing arrays of zero (`0`) dimensions (sometimes called *empties*) results in an empty matrix from MATLAB.

Java Signature	Value Returned from MATLAB
`double[] foo()`	`[]`

Passing Java Empties to MATLAB

When a `null` is passed from Java to MATLAB, it will always be marshaled into `[]` in MATLAB as a zero by zero (`0 x 0`) double. This is independent of the declared input type used in Java. For example, all the following methods can accept `null` as an input value:

```
void foo(String input);
void foo(double[] input);
void foo(double[][] input);
void foo(Double input);
```

And in MATLAB, `null` will be received as:

```
[] i.e. 0x0 double
```

Passing MATLAB Empties to Java

An empty array in MATLAB has at least one zero (`0`) assigned in at least one dimension. For `function a = foo`, for example, any one of the following values is acceptable:

```
 a = [];
 a = ones(0);
 a = ones(0,0);
 a = ones(1,2,0,3);
```

Empty MATLAB data will be returned to Java as `null` for all the above cases.

For example, in Java, the following signatures return `null` when a MATLAB function returns an empty array:

```
double[] foo();
double[][] foo();
Double foo();
```

However, when MATLAB returns an empty array and the return type in Java is a `scalar` primitive (as with `double foo();`, for example) an exception is thrown . :

```
IllegalArgumentException
("An empty MATLAB array cannot be represented by a
  primitive scalar Java type")
```

Boxed Types

Boxed Types are used to wrap opaque C structures.

Java client will perform primitive to boxed type conversion if boxed types are used as return types in the Java method signature.

Java Signature	Value Returned from MATLAB
`Double foo()`	`1.0`

For example, the following method signatures work interchangeably:

```
double[] foo();         Double[] foo();
double[][][] foo();     Double[][][] foo();
```

Signed and Unsigned Types in Java and MATLAB Data Types

Numeric classes in MATLAB include signed and unsigned integers. Java does not have unsigned types.

.NET Client Programming

.NET Client Coding Best Practices

When writing .NET interfaces to invoke MATLAB code, remember these guidelines:

- The method name exposed by the interface *must* match the name of the MATLAB function being deployed. The method must have the same number of inputs and outputs as the MATLAB function.
- The method input and output types must be convertible to and from MATLAB.
- The number of inputs and outputs must be compatible with those supported by MATLAB.
- If you are working with MATLAB structures, remember that the field names are case sensitive and must match in both the MATLAB function and corresponding user-defined .NET type.
- The name of the interface can be any valid .NET name.
- Your code should support exception handling.

.NET Client Prerequisites

Complete these steps to prepare your MATLAB Production Server .NET development environment.

1. Install Microsoft Visual Studio. See http://www.mathworks.com/support/compilers/current_release/ for an up-to-date listing of supported software, including IDEs and Microsoft .NET Frameworks.

2. Verify that you have one generic CTF archive in your server's `auto_deploy` folder for each application interface you plan to create on the client.

   For information about creating a CTF archive with the Deployment Tool, see .

3. Verify that your server `main_config` file is customized to point to where your MCR instance is installed (using `--mcr-root`).

4 Your server must be running in order to perform to the following client example.

Handling Exceptions

You should declare exceptions for the following errors:

For This Error	Use This Method	To Declare This Exception
MATLAB errors	`MATLABException`	`MathWorks.MATLAB.ProductionServer.Client.MWClient.MATLABException`
Transport errors occurring during client-server communication	`WebException`	`System.Net.WebException`

Managing System Resources

A single .NET client connects to one or more servers available at different URLs. Even though users sometimes create multiple instances of `MWHttpClient`, you can use a single instance to communicate with more than one server. The server and client have a one to one relationship at any point in time. The server cannot communicate with multiple clients simultaneously.

Proxy objects, created using an instance of `MWHttpClient`, communicate with the server until the `Dispose` method of that instance is invoked. Therefore, it is important to call the `Dispose` method only when the `MWHttpClient` instance is no longer needed, to reclaim system resources.

Using IDisposable to Release Resources Consumed by MWHttpClient Instances

Call the `Dispose` method (an implementation of `IDisposable`) on unneeded client instances to free native resources, such as open connections created by an instance of `MWHttpClient`.

You call `Dispose` in either of two ways:

- **Call `Dispose` Directly** — Call the method directly on the object whose resources you want to free:

  ```
  client.Dispose();
  ```

- **The `using` keyword** — Implicitly invoke `Dispose` on the`MWHttpClient` instance with the `using` keyword. By doing this, you don't have to explicitly call the `Dispose` method—the .NET Framework handles cleanup for you.

  Following is a code snippet that demonstrates use of the `using` keyword:

  ```
  using (MWClient client = new MWHttpClient(new TestConfigDispose()))
  {
         // Use client to create proxy instances and invoke
         //   MATLAB functions....
  }
  ```

Caution Calling Dispose on instances of `MWClient` closes *all* open sockets bound to the instance.

Configure Client Timeout Value for Connection with a Server

To prevent client and server deadlocks and to ensure client stability, consider setting a timeout parameter when the client is connected with the server and the server becomes unresponsive.

To set a timeout parameter in milliseconds, implement interface `MWHttpClientConfig` in your client code. Use the overloaded constructor of `MWHttpClient`, which takes in an instance of `MWHttpClientConfig`.

For example, to set the client to timeout after no response from the server after 3 minutes (180000 milliseconds), add the following to your client code (as shown in the Magic Square example in).

```
class CustomConfig : MWHttpClientConfig
{
    public int TimeoutMilliSeconds
    {
        get { return 180000; }
```

```
    }
}
```

Note The default timeout parameter is 120000 milliseconds (2 minutes).

Data Conversion for .NET and MATLAB Types

For information regarding supported MATLAB types for client and server marshaling, see

Where to Find the API Documentation

The API doc for the .NET client is installed in *`$MPS_INSTALL`*`/client`.

Preparing Your Microsoft Visual Studio Environment

Before you begin writing the .NET application interface, complete the following steps to prepare your development environment.

Creating a Microsoft Visual Studio Project

1. Open Microsoft Visual Studio.

2. Click **File > New > Project**.

3. In the New Project dialog box, 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. Select the **C# Console Application** template from the **Templates** pane.

4. Type the name of the project in the **Name** field (`MainApp`, for example).

5. Click **OK**. Your `MainApp` source shell is created.

Creating 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, `MainApp`, highlighting it.

2. Right-click `MainApp` 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.

Access Secure Programs Using HTTPS

In this section...

Overview

Connecting to a MATLAB Production Server instance over HTTPS provides a secure channel for executing MATLAB functions. To establish an HTTPS connection with a MATLAB Production Server instance:

1. Ensure that the server is configured to use HTTPS.
2. Install the required credentials on the client system.
3. Configure the client’s .Net environment to use the credentials.
4. Create the program proxy using the program’s `https://` URL.

Configure the Client Environment for SSL

At a minimum the client requires the server’s root CA (Certificate Authority) in one of the application’s certificate stores. If the client needs to connect to a server that requires client-side authentication, it also needs a signed certificate in one of the application’s certificate stores.

Use `makecert` to manage the client’s certificates. For information on using `makecert` see the MSDN documentation.

Establish a Secure Proxy Connection

You can create a secure proxy connection with a MATLAB Production Server instance by using the `https://` URL for the desired program:

```
MWClient client = new MWHttpClient();
Uri secureUri = new Uri("https://host:port/myCTF")
MyCTFProxy sslProxy = client.createProxy<MyCTFProxy>(secureUri);
```

`sslProxy` checks the application's certificate stores to perform the HTTPS server authentication. If the server requests client authentication, the HTTPS handshake will fail because the client does not have a certificate.

Establish a Secure Connection using Client Authentication

To enable your client to connect with a server instance requiring client authentication, you:

1. Provide an implementation of the `MWSSLConfig` interface that returns a valid client certificate collection.

2. Use the `MWHttpClient` constructor that takes an instance of your `MWSSLConfig` implementation to create the connection to the server instance.

3. Create the proxy using the program's `https://` URL.

Implement the `MWSSLConfig` Interface

The `MWSSLConfig` interface has a single property, `ClientCertificates`, of type `X509CertificateCollection`. You need to provide an implementation that returns the client's certificates.

```
public class ClientSSLConfig : MWSSLConfig
{
  public X509CertificateCollection ClientCertificates
  {
    get
    {
      X509Certificate2 clientCert = new X509Certificate2("C:\temp\certifica
      return new X509Certificate2Collection(clientCert);
    }
  }
}
```

Establish the Secure Connection

You create a secure proxy connection with a MATLAB Production Server instance by using the constructor that takes an instance of your `MWSSLConfig` implementation and creating the proxy with the `https://` URL for the desired program:

```
MWClient client = new MWHttpClient(new ClientSSLConfig());
Uri secureUri = new Uri("https://host:port/myCTF")
MyCTFProxy sslProxy = client.createProxy<MyCTFProxy>(secureUri);
```

`sslProxy` will use the local user trust store to perform the HTTPS server authentication. If the server requests client authentication, the client passes the certificates in the collection returned by your implementation of the `MWSSLConfig` interface.

Implementing Advanced Authentication Features

You can perform alternate hostname verification to authenticate servers when the URL hostname does not match the certificate's hostname. You can also perform an extra level of server authorization to ensure that the client only shares data with very specific servers. These extra layers of security are performed using the .Net `ServicePointManager.ServerCertificateValidationCallback` property.

The `ServerCertificateValidationCallback` property is a delegate that is used to process the certificates during the SSL handshake. By default, no delegate is implemented so no custom processing is performed. You can provide an implementation to perform any custom authorization required. For more information see the .Net `ServicePointManager` documentation.

Bond Pricing Tool with GUI for .NET Client

This example shows an application that calculates a bond price from a simple formula.

You run this example by entering the following known values into a simple graphical interface:

- Coupon payment — C
- Number of payments — N
- Interest rate — i
- Value of bond or option at maturity — M

The application calculates price (P) based on the following equation:

```
P = C * ( (1 - (1 + i)^-N) / i ) + M * (1 + i)^-N
```

Objectives

The Bond Pricing Tool demonstrates the following features of MATLAB Production Server:

- Deploying a simple MATLAB function with a fixed number of inputs and a single output
- Deploying a MATLAB function with a simple GUI front-end for data input
- Using `dispose()` to free system resources

Step 1: Write MATLAB Code

Implement the Bond Pricing Tool in MATLAB, by writing the following code. Name the code `pricecalc.m`.

Sample code is available in *MPS_INSTALL*`\client\java\examples\MATLAB`.

```
function price = pricecalc(value_at_maturity, coupon_payment,...
                           interest_rate, num_payments)

    C = coupon_payment;
    N = num_payments;
    i = interest_rate;
    M = value_at_maturity;

    price = C * ( (1 - (1 + i)^-N) / i ) + M * (1 + i)^-N;

end
```

Step 2: Create a Deployable CTF Archive with the Library Compiler App

To create the deployable archive for this example:

1. From MATLAB, select the Library Compiler App.
2. In the **Application Type** list, select **Generic CTF**.
3. In the **Exported Functions** field, add `pricecalc.m`.

   `pricecalc.m` is located in `*MPS_INSTALL*\client\java\examples\BondPricingTool\MATLAB`.
4. Under **Application Information**, change `pricecalc` to `BondTools`.
5. Click **Package**.

   The generated deployable archive, `BondTools.ctf` is located in the `for_redistribution_files_only` of the project's folder.

Step 3: Share the Deployable CTF Archive on a Server

1. Download the MATLAB Compiler Runtime, if needed, at http://www.mathworks.com/products/compiler/mcr. See "MATLAB Compiler Runtime (MCR) Installation" on page 5-14 for more information.
2. Create a server using `mps-new`. See "Server Creation" on page 5-11 for more information.
3. If you have not already done so, specify the location of the MATLAB Compiler Runtime (MCR) to the server by editing the server configuration file, `main_config` and specifying a path for `--mcr-root`. See "Configuration File Customization" on page 5-22 for details.

4 Start the server using `mps-start` and verify it is running with `mps-status`.

5 Copy the `BondTools.ctf` file to the`auto_deploy` folder on the server for hosting. See for complete details.

Step 4: Create the C# Client Code

Create a compatible client interface, defining methods in C# to match MATLAB function `pricecalc.m`, hosted by the server as `BondTools.ctf`, using the guidelines in this section.

When developing your C# code, perform the following tasks, described in the sections that follow. For more information about clients coding basics and best practices, see ".NET Client Coding Best Practices" on page 8-2.

Declare C# Method Signatures Compatible with MATLAB Functions You Deploy

To use the MATLAB functions you defined in "Step 1: Write MATLAB Code" on page 7-8, declare the corresponding C# method signature in the interface `BondTools.cs`:

```
public interface BondTools
    {
        double pricecalc(double faceValue, double couponYield,
                      double interestRate, double numPayments);
    }
```

This interface creates an array of primitive `double` types, corresponding to the MATLAB primitive types (`Double`, in MATLAB, unless explicitly declared) in `pricecalc.m`. A one to one mapping exists between the input arguments in both the MATLAB function and the C# interface The interface specifies compatible type `double`. This compliance between the MATLAB and C# signatures demonstrates the guidelines listed in .

Instantiate MWClient, Create Proxy, and Specify Deployable CTF Archive

In the `ServerBondToolsFactory` class, perform a typical MATLAB Production Server client setup:

1 Instantiate `MWClient` with an instance of `MWHttpClient`:

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

2 Call `createProxy` on the new client instance. Specify port number (`9910`) and the CTF archive name (`BondTools`) the server is hosting in the `auto_deploy` folder:

```
...
  public BondTools newInstance()
        {
            return client.CreateProxy<BondTools>(new Uri("http://localhost:9910/BondTools"));
        }...
```

Use Dispose() Consistently to Free System Resources

This application makes use of the Factory pattern to encapsulate creation of several types of objects.

Any time you create objects—and therefore allocate resources—ensure you free those resources using `Dispose()`.

For example, note that in `ServerBondToolsFactory.cs`, you dispose of the `MWHttpClient` instance you created in "Instantiate MWClient, Create Proxy, and Specify Deployable CTF Archive" on page 8-12 when it is no longer needed.

Additionally, note the `Dispose()` calls to clean up the factories in `BondToolsStubFactory.cs` and `BondTools.cs`.

`Dispose()` is an implementation of `IDisposable`. For more information about using `Dispose()` to free resources, see "Use Dispose() Consistently to Free System Resources" on page 8-13.

Step 5: Build the Client Code and Run the Example

Before you attempt to build and run your client code, ensure that you have done the following:

- Added the `Mathworks.MATLAB.ProductionServer.Client.dll` assembly (*`$MPS_INSTALL`*`\client\net`) as a reference to your Microsoft Visual Studio project.
- Copied your deployable CTF archive to your server's `auto_deploy` folder.
- Modified your server's `main_config` file to point to where your MCR is installed (using `mcr-root`).
- Started your server and verified it is running.

Code Multiple Outputs for C# .NET Client

MATLAB allows users to write functions with multiple outputs. To code multiple outputs in C#, use the `out` keyword.

The following MATLAB code takes multiple inputs (`i1`, `i2`, `i3`) and returns multiple outputs (`o1`, `o2`, `o3`), after performing some checks and calculations.

In this example, the first input and output are of type `double`, and the second input and output are of type `int`. The third input and output are of type `string`.

To deploy this function with MATLAB Production Server software, you need to write a corresponding method interface in C#, using the `out` keyword. Specifying the `out` keyword causes arguments to be passed by reference. When using `out`, ensure both the interface method definition and the calling method explicitly specify the `out` keyword.

The output argument data types listed in your C# interface (referenced with the `out` keyword) must match the output argument data types listed in your MATLAB signature exactly. Therefore, in the C# interface (`MultipleOutputsExample`) and method (`TryMultipleOutputs`) code samples, multiple outputs are listed (with matching specified data types) in the same order as they are listed in your MATLAB function.

MATLAB Function multipleoutputs

```
function [o1 o2 o3] = multipleoutputs(i1, i2, i3)
o1 = modifyinput(i1);
o2 = modifyinput(i2);
o3 = modifyinput(i3);

function out = modifyinput(in)
if( isnumeric(in) )
    out = in*2;
elseif( ischar(in) )
    out = upper(in);
else
    out = in;
end
```

C# Interface MultipleOutputsExample

```
public interface MultipleOutputsExample
{
    void multipleoutputs(out double o1, out int o2, out string o3,
                                     double i1, int i2, string i3);
      }
```

C# Method TryMultipleOutputs

```
public static void TryMultipleOutputs()
{
    MWClient client = new MWHttpClient();
    MultipleOutputsExample mpsexample =
      client.CreateProxy<MultipleOutputsExample>(new Uri("http://localhost:9910/mpsexample"));

    double o1;
    int o2;
    string o3;
    mpsexample.multipleoutputs(out o1, out o2, out o3, 1.2, 10, "hello");
  }
```

After creating a new instance of `MWHttpClient` and a client proxy, variables and the calling method, `multipleoutputs`, are declared.

In the `multipleoutputs` method, values matching each declared types are passed for output (`1.2` for `double`, `10` for `int`, and `hello` for `string`) to `output1`.

Note the following coding best practices illustrated by this example:

- Both the MATLAB function signature and the C# interface method signature use the name `multipleOutputs`. Both MATLAB and C# code are processing three inputs and three outputs.
- MATLAB .NET interface supports direct conversion from C# `double` array to MATLAB `double` array and from C# `string` to MATLAB char array. For more information, see “Data Conversion with C# and MATLAB Types” on page 8-34 and “Conversion Between MATLAB Types and C# Types” on page A-6.

Code Variable-Length Inputs and Outputs for .NET Client

MATLAB Production Server .NET client supports the MATLAB capability of working with variable-length inputs. See the *MATLAB Function Reference* for complete information on varargin and varargout.

Using varargin with .NET Client

You pass MATLAB variable input arguments (varargin) using the params keyword.

For example, consider the MATLAB function varargintest, which takes a variable-length input (varargin)—containing strings and integers—and returns an array of cells (o).

MATLAB Function varargintest

```
function o = varargintest(s1, i2, varargin)

o{1} = s1;
o{2} = i2;
idx = 3;
for i=1:length(varargin)
   o{idx} = varargin{i};
   idx = idx+1;
end
```

The C# interface VarargInTest implements the MATLAB function varargintest.

C# Interface VarargInTest

```
public interface VarargInTest
{
    object[] varargintest(string s, int i, params object[] objArg);
}
```

Since you are sending output to cell arrays in MATLAB, you define a compatible C# array type of object[] in your interface. *objArg* defines number of inputs passed—in this case, two.

The C# method TryVarargin implements VarargінTest, sending two strings and two integers to the deployed MATLAB function, to be returned as a cell array.

C# Method TryVarargin

```
public static void TryVarargin()
{
    MWClient client = new MWHttpClient();
    VararginTest mpsexample =
         client.CreateProxy<VararginTest>(new Uri("http://localhost:9910/mpsexample"));
    object[] vOut = mpsexample.varargintest("test", 20, false, new int[]{1,2,3});
    Console.ReadLine();
}
```

Note the following coding best practices illustrated by this example:

- Both the MATLAB function signature and the C# interface method signature use the name varargintest. Both MATLAB and C# code are processing two variable-length inputs, string and integer.
- MATLAB .NET interface supports direct conversion between MATLAB cell arrays and C# object arrays. See “Data Conversion with C# and MATLAB Types” on page 8-34 and “Conversion Between MATLAB Types and C# Types” on page A-6 for more information.

Using varargout with .NET Client

MATLAB variable output arguments (varargout) are obtained by passing an instance of System.Object[] array. The array is passed with the attribute [varargout], defined in the Mathworks.MATLAB.ProductionServer.Client.dll assembly.

Before passing the System.Object[] instance, initialize the System.Object array instance with the maximum length of the variable in your calling method. The array is limited to one dimension.

For example, consider the MATLAB function varargouttest, which takes one variable-length input (varargin), and returns one variable-length output (varargout), as well as two non-variable-length outputs (out1 and out2).

MATLAB Function varargouttest

```
functionout [out1 out2 varargout] = varargouttest(in1, in2, varargin)

out1 = modifyinput(in1);
out2 =modifyinput(in2);

for i=1:length(varargin)
    varargout{i} = modifyinput(varargin{i});
end

function out = modifyinput(in)
if ( isnumeric(in) )
    out = in*2;
elseif ( ischar(in) )
    out = upper(in);
elseif ( islogical(in) )
    out = ~in;
else
    out = in;
end
```

Implement MATLAB function varargouttest with the C# interface VarargoutTest.

In the interface method varargouttest, you define multiple non-variable-length outputs (o1 and o2, using the out keyword, described in “Code Multiple Outputs for C# .NET Client” on page 8-15), a double input (in1) and a string input (in2).

You pass the variable-length output (o3) using a single-dimensional array (object[] with attribute [varargout]), an instance of System.Object[].

As with “Using varargin with .NET Client” on page 8-17, you use the params keyword to pass the variable-length input.

C# Interface VarargoutTest

```
public interface VarargOutTest
{
    void varargouttest(out double o1, out string o2, double in1, string in2
        [varargout] object[] o3, params object[] varargIn);
}
```

In the calling method `TryVarargout`, note that both the type and length of the variable output (`varargOut`) are being passed (`(short)12`).

C# Method TryVarargout

Note Ensure that you initialize `varargOut` to the appropriate length before passing it as input to the method `varargouttest`.

```
public static void TryVarargout()
{
    MWClient client = new MWHttpClient();
    VarargOutTest mpsexample =
       client.CreateProxy<VarargOutTest>(new Uri("http://localhost:9910/mpsexample"));

    object[] varargOut = new object[3]; // get all 3 outputs
    double o1;
    string o2;
    mpsexample.varargouttest(out o1, out o2, 1.2, "hello",
                       varargOut, true, (short)12, "test");

    varargOut = new object[2];  // only get 2 outputs
    double o11;
    string o22;
    mpsexample.varargouttest(out o11, out o22, 1.2, "hello",
                         varargOut, true, (short)12, "test");
}
```

Note the following coding best practices illustrated by this example:

- Both the MATLAB function signature and the C# interface method signature use the name `varargouttest`. Both MATLAB and C# code are processing a variable-length input, a variable-length output, and two multiple non-variable-length outputs.

- MATLAB .NET interface supports direct conversion between MATLAB cell arrays and C# object arrays. See “Data Conversion with C# and MATLAB Types” on page 8-34 and “Conversion Between MATLAB Types and C# Types” on page A-6 for more information.

Marshal MATLAB Structures (structs) in C#

Structures (or *structs*) are MATLAB arrays with elements accessed by textual field designators.

Structs consist of data containers, called *fields*. Each field stores an array of some MATLAB data type. Every field has a unique name.

Creating a MATLAB Structure

MATLAB structures are ordered lists of name-value pairs. You represent them in C# by defining a .NET struct or class, as long as it has `public` fields or properties corresponding to the MATLAB structure. A field or property in a .NET struct or class can have a value convertible to and from any MATLAB data type, including a cell array or another structure. The examples in this article use both .NET `structs` and classes.

In MATLAB, a student structure containing `name`, `score`, and `grade`, is created as follows:

```
S.name = 'Ed Plum';
S.score = 83;
S.grade = 'B+'
```

This code creates a scalar structure (`S`) with three fields:

```
S =
    name: 'Ed Plum'
    score: 83
    grade: 'B+'
```

A multidimensional structure array can be created by inserting additional elements. A structure array of dimensions `(1,3)` is created. For example:

```
S(2).name = 'Tony Miller';
S(2).score = 91;
S(2).grade = 'A-';

S(3).name = 'Mark Jones';
S(3).score = 85;
S(3).grade = 'A-';
```

Using .NET Structs and Classes

You create .NET structs and classes to marshal data to and from MATLAB structures.

The .NET `struct Student` is an example of a .NET `struct` that is marshaling .NET types as inputs to MATLAB function, such as `sortstudents`, using `public` *fields and properties*. Note the publicly declared field `name`, and the properties `grade` and `score`.

In addition to using a .NET `struct`, Please note the following:

- `Student` can also be defined as a class.
- Even though in this example a combination of `public` fields and properties is used, you can also use *only* fields or properties.

.NET Struct Student

```
public struct Student
{
    public string name;
    private string gr;
    private int sc;

    public string grade
    {
        get { return gr; }
        set { gr = value; }
    }

    public int score
    {
        get { return sc; }
        set { sc = value; }
    }

    public override string ToString()
    {
        return name + " : " + grade + " : " + score;
    }
}
```

Note Note that this example uses the ToString for convenience. It is not required for marshaling.

The C# class SimpleStruct uses public readable properties as input to MATLAB, and uses a public constructor when marshaling as output from MATLAB.

When this class is passed as input to a MATLAB function, it results in a MATLAB struct with fields Field1 and Field2, which are defined as public readable properties. When a MATLAB struct with field names Field1 and Field2 is passed from MATLAB, it is used as the target .NET type (string and double, respectively) because it has a constructor with input parameters Field1 and Field2.

C# Class SimpleStruct

```
public class SimpleStructExample
 {
     private string f1;
     private double f2;

     public SimpleStruct(string Field1, double Field2)
     {
         f1 = Field1;
         f2 = Field2;
     }

     public string Field1
     {
         get
         {
             return f1;
         }
     }
     public double Field2
     {
         get
         {
             return f2;
         }
     }
 }
```

MATLAB function `sortstudents` takes in an array of `student` structures and sorts the input array in ascending order by score of each student. Each element in the `struct` array represents different information about a `student`.

The C# interface `StudentSorter` and method `sortstudents` is provided to show equivalent functionality in C#.

Your .NET structs and classes must adhere to specific requirements, based on both the level of scoping (fields and properties as opposed to constructor, for example) and whether you are marshaling .NET types to or from a MATLAB structure. See for details.

MATLAB Function sortstudents

```
function sorted = sortstudents(unsorted)
% Receive a vector of students as input
% Get scores of all the students
scores = {unsorted.score};
% Convert the cell array containing scores into a numeric array or doubles
scores = cell2mat(scores);
% Sort the scores array
[s i] = sort(scores);
% Sort the students array based on the sorted scores array
sorted = unsorted(i);
```

Note Even though this example only uses the `scores` field of the input structure, you can also work with `name` and `grade` fields in a similar manner.

The .NET interface StudentSorter, with method sortstudents, uses the previously defined .NET Student struct for inputs and outputs. When marshaling structs for input and output in .NET, the Student struct or class must be included in the MWStructureList attribute. Please refer the documentation for this custom attribute in the API documentation, located in *$MPS_INSTALL*/client.

C# Interface StudentSorter

```
public interface StudentSorter {
    [MWStructureList(typeof(Student))]
    Student[] sortstudents(Student[] students);
}
```

C# Class ClientExample

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

namespace MPS
{
    public interface StudentSorter
    {
        [MWStructureList(typeof(Student))]
        Student[] sortstudents(Student[] students);
    }

    class ClientExample
    {
        static void Main(string[] args)
        {
            MWClient client = null;

            try
            {
                client = new MWHttpClient();
                StudentSorter mtsexample =
                    client.CreateProxy(new Uri("http://test-machine:9910/scoresorter"));
                Student s1 = new Student();
                s1.name = "Tony Miller";
                s1.score = 90;
                s1.grade = "A";

                Student s2 = new Student();
                s2.name = "Ed Plum";
                s2.score = 80;
                s2.grade = "B+";

                Student s3 = new Student();
                s3.name = "Mark Jones";
                s3.score = 85;
                s3.grade = "A-";

                Student[] unsorted = new Student[] { s1, s2, s3 };

                Console.WriteLine("Unsorted list of students :");
                foreach (Student st in unsorted)
                {
                    Console.WriteLine(st);
                }

                Console.WriteLine();
                Console.WriteLine("Sorted list of students :");

                Student[] sorted = mtsexample.sortstudents(unsorted);

                foreach (Student st in sorted)
                {
                    Console.WriteLine(st);
                }

            }
            catch (WebException ex)
            {
                HttpWebResponse response = (HttpWebResponse)ex.Response;
                if (response != null)
                {
                    Console.WriteLine("Status code : " +
                                           response.StatusCode);
                    Console.WriteLine("Status description : " +
                                    response.StatusDescription);
                }
                else
                {
                    Console.WriteLine("No response received in
                             WebException with status : " + ex.Status);
                }

            }
            catch (MATLABException ex)
            {
                Console.WriteLine("MATLAB error thrown : ");
                Console.WriteLine(ex.MATLABIdentifier);
                Console.WriteLine(ex.MATLABStackTraceString);
            }
            finally
            {
                if (client != null)
                {
                    client.Dispose();
                }
            }
        }
    }
}
```

When you run the application, the following output is generated:

```
Unsorted list of students :
Tony Miller : A : 90
Ed Plum : B+ : 80
Mark Jones : A- : 85

Sorted list of students :
Ed Plum : B+ : 80
Mark Jones : A- : 85
Tony Miller : A : 90
Press any key to continue . . .
```

Using Attributes

In addition to using the techniques described in “Using .NET Structs and Classes” on page 8-23, attributes also provide versatile ways to marshal .NET types to and from MATLAB structures.

The MATLAB Production Server-defined attribute `MWStructureList` can be scoped at field, property, method, or interface level..

In the following example, a MATLAB function takes a `cell` array (vector) as input containing various MATLAB `struct` data types and returns a `cell` array (vector) as output containing modified versions of the input `structs`.

MATLAB Function outcell

```
function outCell = modifyinput(inCell)
```

Define the `cell` array using two .NET struct types:

.NET struct Types Struct1 and Struct2

```
public struct Struct1{
    ...
    ...
}
public struct Struct2{
    ...
    ...
}
```

Without using the `MWStructureList` attribute, the C# method signature in the interface `StructExample`, is as follows:

```
public interface StructExample
{
    public object[] modifyinput(object[] cellArrayWithStructs);
}
```

Note that this signature, as written, provides no information about the structure types that `cellArrayWithStructs` include at run-time. By using the MWStructureList attribute, however, you define those types directly in the method signature:

```
public interface StructExample
{
    [MWStructureList(typeof(Struct1), typeof(Struct2))]
    public object[] modifyinput(object[] cellArrayWithStructs);
}
```

The MWStructureList attribute can be scoped at:

- “Method Attributes” on page 8-30
- “Interface Attributes” on page 8-31
- “Fields and Property Attributes” on page 8-31

i

Method Attributes

In this example, the attribute MWStructureList is used as a *method attribute* for marshaling both the input and output types.

```
public interface StructExample
{
    [MWStructureList(typeof(Struct1), typeof(Struct2))]
    public object[] modifyinput(object[] cellArrayWithStructs);
}
```

In this example, struct types Struct1 and Struct2 are *not* exposed to method modifyinputNew because modifyinputNew is a separate method signature

```
public interface StructExample
{
    [MWStructureList(typeof(Struct1), typeof(Struct2))]
    public object[] modifyinput(object[] cellArrayWithStructs);
    public object[] modifyinputNew(object[] cellArrayWithStructs);
}
```

Interface Attributes

When used at an interface level, an attribute is shared by all the methods of the interface.

In the following example, both modifyinput and modifyinputNew methods share the interface attribute MWStructureList because the attribute is defined prior to the interface declaration.

```
[MWStructureList(typeof(Struct1), typeof(Struct2))]
public interface StructExample
{
    public object[] modifyinput(object[] cellArrayWithStructs);
    public object[] modifyinputNew(object[] cellArrayWithStructs);
}
```

Fields and Property Attributes

Write the interface using public fields or public properties.

You can represent this type of .NET struct in three ways using fields and properties:

- *At the field:*

  Using public field and the MWStructureList attribute:

  ```
  public struct StructWithinStruct
  {
      [MWStructureList(typeof(Struct1), typeof(Struct2))]
      public object[] cellArrayWithStructs;
  }
  ```

- *At the property, for both get and set methods:*

  Using public properties and the MWStructureList attribute:

  ```
  public struct StructWithinStruct
  {
      private object[] arr;

      [MWStructureList(typeof(Struct1), typeof(Struct2))]
  ```

```
    public object[] cellArrayWithStructs
    {
        get
        {
            return arr;
        }

        set
        {
            arr = value;
        }
    }
}
```

- *At the property, for both or either* `get` *or* `set` *methods, depending on whether this struct will be used as an input to MATLAB or an output from MATLAB:*

  ```
  public struct StructWithinStruct
  {
      private object[] arr;

      public object[] cellArrayWithStructs
      {
          [MWStructureList(typeof(Struct1), typeof(Struct2))]
          get
          {
              return arr;
          }

          [MWStructureList(typeof(Struct1), typeof(Struct2))]
          set
          {
              arr = value;
          }
      }
  }
  ```

Note The last two examples, which show attributes used at the property, produce the same result.

Data Conversion with C# and MATLAB Types

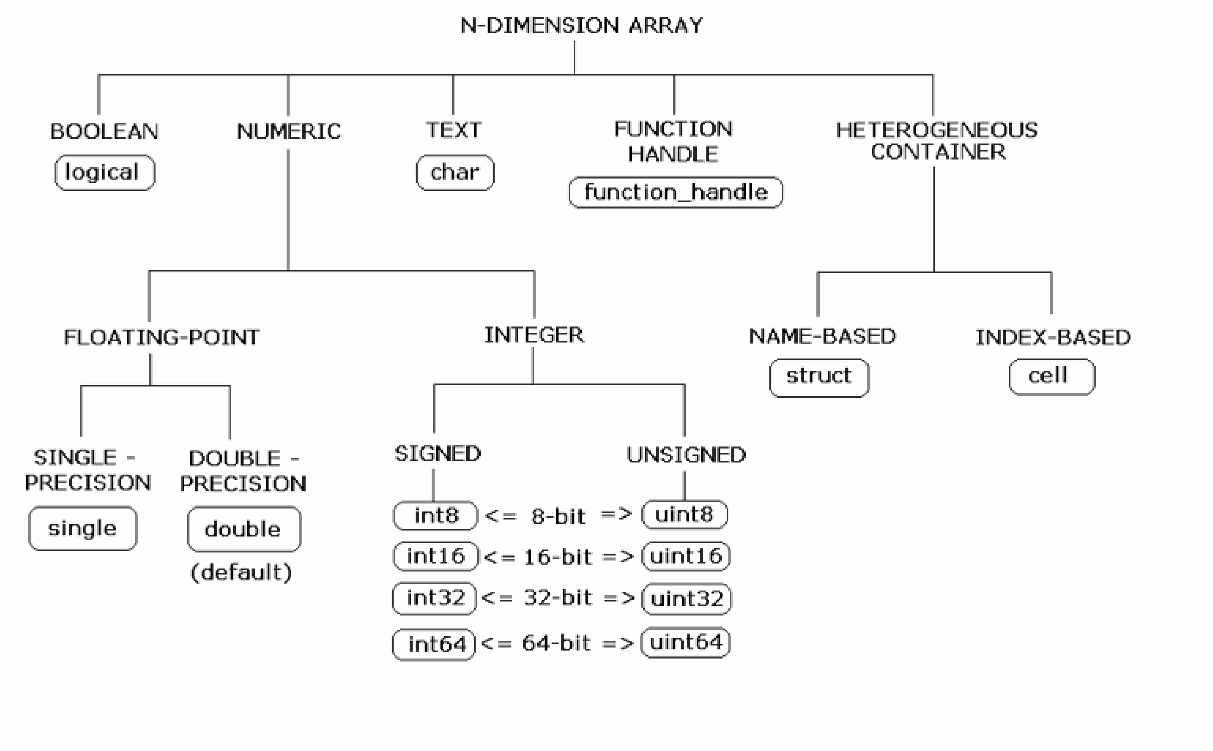
When the .NET client invokes a MATLAB function through a request and receives a result in the response, data conversion takes place between MATLAB types and C# types.

Working with MATLAB Data Types

There are many data types, or classes, that you can work with in MATLAB. Each of these classes is in the form of a matrix or array. You can build matrices and arrays of floating-point and integer data, characters and strings, and logical true and false states. Structures and cell arrays provide a way to store dissimilar types of data in the same array.

All of the fundamental MATLAB classes are circled in the diagram Fundamental MATLAB Data Classes on page 8-35.

Note Function Handles are not supported by MATLAB Production Server.



Fundamental MATLAB Data Classes

Each MATLAB data type has a specific equivalent in C#. Detailed descriptions of these one-to-one relationships are defined in "Conversion Between MATLAB Types and C# Types" on page A-6 in Appendix A, "Data Conversion Rules".

Scalar Numeric Type Coercion

Scalar numeric MATLAB types can be assigned to multiple .NET numeric types as long as there is no loss of data or precision.

The main exception to this rule is that MATLAB `double` scalar data can be mapped into any .NET numeric type. Because `double` is the default numeric type in MATLAB, this exception provides more flexibility to the users of MATLAB Production Server .NET client API.

MATLAB to .NET Numeric Type Compatibility on page 8-36 describes the type compatibility for scalar numeric coercion.

MATLAB to .NET Numeric Type Compatibility

MATLAB Type	Java Types
`uint8`	`System.Int16`, `System.UInt16`, `System.Int32`, `System.UInt32`, `System.Int64`, `System.UInt64`, `System.Single`, `System.Double`
`int8`	`System.Int16`, `System.Int32`, `System.Int64`, `System.Single`, `System.Double`
`uint16`	`System.Int32`, `System.UInt32`, `System.Int64`, `System.UInt64`, `System.Single`, `System.Double`
`int16`	`System.Int32`, `System.Int64`, `System.Single`, `System.Double`
`uint32`	`System.Int64`, `System.UInt64`, `System.Single`, `System.Double`
`int32`	`System.Int64`, `System.Single`, `System.Double`
`uint64`	`System.Single`, `System.Double`
`int64`	`System.Single`, `System.Double`
`single`	`System.Double`
`double`	`System.SByte`, `System.Byte`, `System.Int16`, `System.UInt16`, `System.Int32`, `System.UInt32`, `System.Int64`, `System.UInt64`, `System.Single`

Dimension Coercion

In MATLAB, dimensionality is an attribute of the fundamental types and does not add to the number of types as it does in .NET.

In C#, `double`, `double[]` and `double[,]` are three different data types. In MATLAB, there is only a `double` data type and possibly a scalar instance, a vector instance, or a multi-dimensional instance.

C# Signature	Value Returned from MATLAB
`double[,,] foo()`	`ones(1,2,3)`

How you define your MATLAB function and corresponding C# method signature determines if your output data will be coerced, using padding or truncation.

This coercion is performed automatically for you. This section describes the rules followed for padding and truncation.

Note Multidimensional arrays of C# types are supported. Jagged arrays are not supported.

Padding

When a C# method's return type has a greater number of dimensions than MATLAB's, MATLAB's dimensions are padded with ones (1s) to match the required number of output dimensions in C#.

The following tables provide examples of how padding is performed for you:

How Your C# Method Return Type is Padded

MATLAB Function	C# Method Signature	When Dimensions in MATLAB are:	And Dimensions in C# are:

`function a = foo`
`a = ones(2,3);`
	`double[,,,] foo()`	`size(a)` is `[2,3]`	Array will be returned as size `2,3,1,1`

Truncation

When a C# method's return type has fewer dimensions than MATLAB's, MATLAB's dimensions are truncated to match the required number of output dimensions in C#. This is only possible when extra dimensions for MATLAB array have values of ones (1s) only.

To compute appropriate number of dimensions in C#, excess ones are truncated, in this order:

1. From the end of the array
2. From the array's beginning
3. From the middle of the array (scanning front-to-back).

The following tables provide examples of how truncation is performed for you:

How MATLAB Truncates Your C# Method Return Type

MATLAB Function	C# Method Signature	When Dimensions in MATLAB are:	And Dimensions in C# are:

`function a = foo`
`a = ones(1,2,1,1,3,1);`
	`double[,] foo()`	`size(a) is [1,2,1,1,3,1]`	Array will be returned as size `2,3`

Following are some examples of dimension shortening using the `double numeric` type:

Truncating Dimensions in MATLAB and C# Data Conversion

MATLAB Array Dimensions	Declared Output C# Type	Output C# Dimensions
`1 x 1`	`double`	0 (scalar)
`2 x 1`	`double[]`	`2`
`1 x 2`	`double[]`	`2`
`2 x 3 x 1`	`double[,]`	`2 x 3`
`1 x 3 x 4`	`double[,]`	`3 x 4`
`1 x 3 x 4 x 1 x 1`	`double[,,]`	`1 x 3 x 4`
`1 x 3 x 1 x 1 x 2 x 1 x 4 x 1`	`double[,,,]`	`3 x 2 x 1 x 4`

Empty (Zero) Dimensions

Passing C# Empties to MATLAB

When a null is passed from C# to MATLAB, it will always be marshaled into [] in MATLAB as a zero by zero (0 x 0) double. This is independent of the declared input type used in C#. For example, all the following methods can accept null as an input value:

```
void foo(String input);
void foo(double[] input);
void foo(double[,] input);
```

And in MATLAB, null will be received as:

```
[] i.e. 0x0 double
```

Passing MATLAB Empties to C#

An empty array in MATLAB has at least one zero (0) assigned in at least one dimension. For function a = foo, for example, any one of the following values is acceptable:

```
 a = [];
 a = ones(0);
 a = ones(0,0);
 a = ones(1,2,0,3);
```

Empty MATLAB data is returned to C# as null for all the above cases.

For example, in C#, the following signatures return null when a MATLAB function returns an empty array:

```
double[] foo();
double[,] foo();
```

9

Commands — Alphabetical List

deploytool

Purpose

Compile and package functions for external deployment

Syntax

`deploytool` [-win32] [[[-build] | [-project]]*`project_name`*]

Description

`deploytool` opens the MATLAB Compiler app.

`deploytool project_name` opens the MATLAB Compiler app with the project preloaded.

`deploytool -build project_name` runs the MATLAB Compiler to build the specified project. The installer is not generated.

`deploytool -package project_name` runs the MATLAB Compiler to build and package the specified project. The installer is generated.

`deploytool -win32` instructs the compiler to build a 32-bit application on a 64-bit system when the following are true:

- You use the same MATLAB installation root (*`matlabroot`*) for both 32-bit and 64-bit versions of MATLAB.
- You are running from a Windows command line (not a MATLAB command line).

Input Arguments

project_name - name of the project to be compiled

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

Purpose

Compile MATLAB functions for deployment

Syntax

mcc {-e} | {-m} [-a *filename*]... [-B *filename*[:*arg*]...] [-C] [-d *outFolder*] [-f *filename*] [-g] [-I *directory*]... [-K] [-M *string*] [-N] [-o *filename*] [-p *path*]... [-R *option*] [-v] [-w *option*[:*msg*]] [-win32] [-Y *filename*] *mfilename*

mcc -l [-a *filename*]... [-B *filename*[:*arg*]...] [-C] [-d *outFolder*] [-f *filename*] [-g] [-I *directory*]... [-K] [-M *string*] [-N] [-o *filename*] [-p *path*]... [-R *option*] [-v] [-w *option*[:*msg*]] [-win32] [-Y *filename*] *mfilename*...

mcc -c [-a *filename*]... [-B *filename*[:*arg*]...] [-C] [-d *outFolder*] [-f *filename*] [-g] [-I *directory*]... [-K] [-M *string*] [-N] [-o *filename*] [-p *path*]... [-R *option*] [-v] [-w *option*[:*msg*]] [-win32] [-Y *filename*] *mfilename*...

mcc -W cpplib:*component_name* -T link:lib [-a *filename*]... [-B *filename*[:*arg*]...] [-C] [-d *outFolder*] [-f *filename*] [-g] [-I *directory*]... [-K] [-M *string*] [-N] [-o *filename*] [-p *path*]... [-R *option*] [-S] [-v] [-w *option*[:*msg*]] [-win32] [-Y *filename*] *mfilename*...

mcc -W dotnet:*component_name*,[*className*], [*framework_version*], *security*, *remote_type* -T link:lib [-a *filename*]... [-B *filename*[:*arg*]...] [-C] [-d *outFolder*] [-f *filename*] [-I *directory*]... [-K] [-M *string*] [-N] [-p *path*]... [-R *option*] [-S] [-v] [-w *option*[:*msg*]] [-win32] [-Y *filename*] *mfilename*... [class{*className*:[m*filename*]...}]...

mcc -W excel:*component_name*,[*className*], [*version*] -T link:lib [-a *filename*]... [-b] [-B *filename*[:*arg*]...] [-C] [-d *outFolder*] [-f *filename*] [-I *directory*]... [-K] [-M *string*] [-N] [-p *path*]... [-R *option*] [-u] [-v] [-w *option*[:*msg*]] [-win32] [-Y *filename*] *mfilename*...

mcc -W java:*packageName*,[*className*] [-a *filename*]... [-b] [-B *filename*[:*arg*]...] [-C] [-d *outFolder*] [-f *filename*] [-I *directory*]... [-K] [-M *string*] [-N] [-p *path*]... [-R *option*] [-S] [-v] [-w *option*[:*msg*]] [-win32] [-Y *filename*m] *filename*... [class{*className*:[m*filename*]...}]...

`mcc` -W CTF:*component_name* [-a *filename*]... [-b] [-B *filename*[:*arg*]...] [-d *outFolder*] [-f *filename*] [-I *directory*]... [-K] [-M *string*] [-N] [-p *path*]... [-R *option*] [-S] [-v] [-w *option*[:*msg*]] [-win32] [-Y *filename*m] *filename*... [class{*className*:[m*filename*]...}]...

`mcc` -?

Description

`mcc -m` *`mfilename`* compiles the function into a standalone application.

This is equivalent to `-W main -T link:exe`.

`mcc -e` *`mfilename`* compiles the function into a standalone application that does not open an MS-DOS® command window.

This is equivalent to `-W WinMain -T link:exe`.

`mcc -l` *`mfilename`*`...` compiles the listed functions into a C shared library and generates C wrapper code for integration with other applications.

This is equivalent to `-W lib:`*`libname`*` -T link:lib`.

`mcc -c` *`mfilename`*`...` generates C wrapper code for the listed functions.

This is equivalent to `-W lib:`*`libname`*` -T codegen`.

`mcc -W cpplib:`*`component_name`*` -T link:lib `*`mfilename`*`...` compiles the listed functions into a C++ shared library and generates C++ wrapper code for integration with other applications.

`mcc -W dotnet:`*`component_name`*`,`*`className`*`,`*`framework_version`*`,` *`security`*`,`*`remote_type`*` -T link:lib `*`mfilename`*`...` creates a .NET component from the specified files.

- `component_name` — Specifies the name of the component and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- `className` — Specifies the name of the .NET class to be created.
- `framework_version` — Specifies the version of the Microsoft .NET Framework you want to use to compile the component. Specify either:
  - `0.0` — Use the latest supported version on the target machine.
  - `version_major.version_minor` — Use a specific version of the framework.

  Features are often version-specific. Consult the documentation for the feature you are implementing to get the Microsoft .NET Framework version requirements.
- `security` — Specifies whether the component to be created is a private assembly or a shared assembly.
  - To create a private assembly, specify `Private`.
  - To create a shared assembly, specify the full path to the encryption key file used to sign the assembly.
- `remote_type` — Specifies the remoting type of the component. Values are `remote` and `local`.

By default, the compiler generates a single class with a method for each function specified on the command line. You can instruct the compiler to create multiple classes using `class{className:mfilename...}....` `className` specifies the name of the class to create using `mfilename`.

`mcc -W excel:component_name,className, version -T link:lib mfilename...` creates a Microsoft Excel component from the specified files.

- `component_name` — Specifies the name of the component and its namespace, which is a period-separated list, such as `companyname.groupname.component`.

- *className* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the *component_name* as the default.
- *version* — Specifies the version of the component specified as *major.minor*.
  - *major* — Specifies the major version number. If you do not specify a version number, `mcc` uses the latest version.
  - *minor* — Specifies the minor version number. If you do not specify a version number, `mcc` uses the latest version.

`mcc -W java:`*packageName*`,`*className* *mfilename*`...` creates a Java package from the specified files.

- *packageName* — Specifies the name of the Java package and its namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *className* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the last item in *packageName*.

By default, the compiler generates a single class with a method for each function specified on the command line. You can instruct the compiler to create multiple classes using `class{`*className*:*mfilename*`...}....` *className* specifies the name of the class to create using *mfilename*.

`mcc -W CTF:`*component_name* instructs the compiler to create a deployable CTF archive that is deployable in a MATLAB Production Server instance.

`mcc -?` displays help.

Tip You can issue the `mcc` command either from the MATLAB command prompt or the DOS or UNIX® command line.

Options

-a Add to Archive

Add a file to the CTF archive using

```
-a filename
```

to specify a file to be directly added to the CTF archive. Multiple `-a` options are permitted. MATLAB Compiler looks for these files on the MATLAB path, so specifying the full path name is optional. These files are not passed to `mbuild`, so you can include files such as data files.

If only a folder name is included with the `-a` option, the entire contents of that folder are added recursively to the CTF archive. For example:

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

In this example, `testdir` is a folder in the current working folder. All files in `testdir`, as well as all files in subfolders of `testdir`, are added to the CTF archive, and the folder subtree in `testdir` is preserved in the CTF archive.

If a wildcard pattern is included in the file name, only the files in the folder that match the pattern are added to the CTF archive and subfolders of the given path are not processed recursively. For example:

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

In this example, all files in `./testdir` are added to the CTF archive and subfolders under `./testdir` are not processed recursively.

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

In this example, all files with the extension `.m` under `./testdir` are added to the CTF archive and subfolders of `./testdir` are not processed recursively.

All files added to the CTF archive using `-a` (including those that match a wildcard pattern or appear under a folder specified using `-a`) that do not appear on the MATLAB path at the time of compilation causes a path entry to be added to the deployed application's run-time path so that they appear on the path when the deployed application or component executes.

When files are included, the absolute path for the DLL and header files is changed. The files are placed in the `.\exe_mcr\` folder when the CTF file is expanded. The file is not placed in the local folder. This folder is created from the CTF file the first time the EXE file is executed. The `isdeployed` function is provided to help you accommodate this difference in deployed mode.

The `-a` switch also creates a `.auth` file for authorization purposes. It ensures that the executable looks for the DLL- and H-files in the *`exe_mcr\exe`* folder.

Caution

If you use the `-a` flag to include a file that is not on the MATLAB path, the folder containing the file is added to the MATLAB dependency analysis path. As a result, other files from that folder might be included in the compiled application.

Note Currently, `*` is the only supported wildcard.

Note If the `-a` flag is used to include custom Java classes, standalone applications work without any need to change the `classpath` as long as the Java class is not a member of a package. The same applies for JAR files. However, if the class being added is a member of a package, the MATLAB code needs to make an appropriate call to `javaaddpath` to update the `classpath` with the parent folder of the package.

-b Generate Excel Compatible Formula Function

Generate a Visual Basic® file (`.bas`) containing the Microsoft Excel Formula Function interface to the COM object generated by MATLAB Compiler. When imported into the workbook Visual Basic code, this code allows the MATLAB function to be seen as a cell formula function. This option requires MATLAB Builder™ EX.

-B Specify Bundle File

Replace the file on the mcc command line with the contents of the specified file. Use

```
-B filename[:<a1>,<a2>,...,<an>]
```

The bundle file filename should contain only mcc command-line options and corresponding arguments and/or other file names. The file might contain other -B options. A bundle file can include replacement parameters for Compiler options that accept names and version numbers. See for a list of the bundle files included with MATLAB Compiler.

-C Do Not Embed CTF Archive by Default

Override automatically embedding the CTF archive in C/C++ and `main`/`Winmain` shared libraries and standalone binaries by default. See for more information.

-d Specified Folder for Output

Place output in a specified folder. Use

```
-d outFolder
```

to direct the generated files to *`outFolder`*.

-f Specified Options File

Override the default options file with the specified options file. Use

```
-f filename
```

to specify `filename` as the options file when calling `mbuild`. This option lets you use different ANSI compilers for different invocations of MATLAB Compiler. This option is a direct pass-through to the `mbuild` script.

Note MathWorks recommends that you use `mbuild -setup`.

-g Generate Debugging Information

Include debugging symbol information for the C/C++ code generated by MATLAB Compiler. It also causes `mbuild` to pass appropriate debugging flags to the system C/C++ compiler. The debug option lets you backtrace up to the point where you can identify if the failure occurred in the initialization of MCR, the function call, or the termination routine. This option does not let you debug your MATLAB files with a C/C++ debugger.

-G Debug Only

Same as `-g`.

-I Add Folder to Include Path

Add a new folder path to the list of included folders. Each `-I` option adds a folder to the beginning of the list of paths to search. For example,

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

sets up the search path so that `directory1` is searched first for MATLAB files, followed by `directory2`. This option is important for standalone compilation where the MATLAB path is not available.

-K Preserve Partial Output Files

Direct `mcc` to not delete output files if the compilation ends prematurely, due to error.

The default behavior of `mcc` is to dispose of any partial output if the command fails to execute successfully.

-M Direct Pass Through

Define compile-time options. Use

```
-M string
```

to pass `string` directly to the `mbuild` script. This provides a useful mechanism for defining compile-time options, e.g., `-M "-Dmacro=value"`.

Note Multiple `-M` options do not accumulate; only the rightmost `-M` option is used.

-N Clear Path

Passing `-N` effectively clears the path of all folders except the following core folders (this list is subject to change over time):

- *`matlabroot`*`\toolbox\matlab`
- *`matlabroot`*`\toolbox\local`
- *`matlabroot`*`\toolbox\compiler\deploy`

It also retains all subfolders of the above list that appear on the MATLAB path at compile time. Including `-N` on the command line lets you replace folders from the original path, while retaining the relative ordering of the included folders. All subfolders of the included folders that appear on the original path are also included. In addition, the `-N` option retains all folders that you included on the path that are not under *`matlabroot`*`\toolbox`.

-o Specify Output Name

Specify the name of the final executable (standalone applications only). Use

```
-o outputfile
```

to name the final executable output of MATLAB Compiler. A suitable, possibly platform-dependent, extension is added to the specified name (e.g., `.exe` for Windows standalone applications).

-p Add Folder to Path

Use in conjunction with the required option `-N` to add specific folders (and subfolders) under *`matlabroot`*`\toolbox` to the compilation MATLAB path in an order sensitive way. Use the syntax

```
-N -p directory
```

where `directory` is the folder to be included. If `directory` is not an absolute path, it is assumed to be under the current working folder. The rules for how these folders are included follow.

- If a folder is included with `-p` that is on the original MATLAB path, the folder and all its subfolders that appear on the original path are added to the compilation path in an order-sensitive context.
- If a folder is included with `-p` that is not on the original MATLAB path, that folder is not included in the compilation. (You can use `-I` to add it.)

If a path is added with the `-I` option while this feature is active (`-N` has been passed) and it is already on the MATLAB path, it is added in the order-sensitive context as if it were included with `-p`. Otherwise, the folder is added to the head of the path, as it normally would be with `-I`.

-R Run-Time

Provide MCR run-time options. Use the syntax

```
-R option
```

to provide one of these run-time options.

Option	Description
`-logfile` *`filename`*	Specify a log file name.
`-nodisplay`	Suppress the MATLAB `nodisplay` run-time warning.
`-nojvm`	Do not use the Java Virtual Machine (JVM).
`-startmsg`	Customizable user message displayed at MCR initialization time. See .
`-completemsg`	Customizable user message displayed when MCR initialization is complete. See .

See for information about using `mcc -R` with initialization messages.

Note The `-R` option is available only for standalone applications. To override MCR options in the other MATLAB Compiler targets, use the `mclInitializeApplication` and `mclTerminateApplication` functions. For more information on these functions, see .

Caution

When running on Mac OS X, if `-nodisplay` is used as one of the options included in `mclInitializeApplication`, then the call to `mclInitializeApplication` must occur before calling `mclRunMain`.

-S Create Singleton MCR

Create a singleton MCR.

The standard behavior for the MCR is that every instance of a class gets its own base workspace. In a singleton MCR, all instances of a class share the same base workspace.

-T Specify Target Stage

Specify the output target phase and type.

Use the syntax `-T` *`target`* to define the output type. Target values are as follow.

Target	Description
`codegen`	Generate a C/C++ wrapper file. The default is `codegen`.
`compile:exe`	Same as `codegen` plus compiles C/C++ files to object form suitable for linking into a standalone application.
`compile:lib`	Same as `codegen` plus compiles C/C++ files to object form suitable for linking into a shared library/DLL.
`link:exe`	Same as `compile:exe` plus links object files into a standalone application.
`link:lib`	Same as `compile:lib` plus links object files into a shared library/DLL.

-u Register COM Component for the Current User

Register COM component for the current user only on the development machine. The argument applies only for generic COM component and Microsoft Excel add-in targets only.

-v Verbose

Display the compilation steps, including:

- MATLAB Compiler version number
- The source file names as they are processed
- The names of the generated output files as they are created
- The invocation of mbuild

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

-w Warning Messages

Display warning messages. Use the syntax

```
-w option [:<msg>]
```

to control the display of warnings. This table lists the syntaxes.

Syntax	Description
`-w list`	Generate a table that maps `<string>` to warning message for use with `enable`, `disable`, and `error`. , lists the same information.
`-w enable`	Enable complete warnings.
`-w disable[:<string>]`	Disable specific warnings associated with `<string>`. , lists the `<string>` values. Omit the optional `<string>` to apply the `disable` action to all warnings.
`-w enable[:<string>]`	Enable specific warnings associated with `<string>`. , lists the `<string>` values. Omit the optional `<string>` to apply the `enable` action to all warnings.
`-w error[:<string>]`	Treat specific warnings associated with `<string>` as an error. Omit the optional `<string>` to apply the `error` action to all warnings.

Syntax	Description
`-w off[:<string>] [<filename>]`	Turn warnings off for specific error messages defined by `<string>`. You can also narrow scope by specifying warnings be turned off when generated by specific `<filename>`s.
`-w on[:<string>] [<filename>]`	Turn warnings on for specific error messages defined by `<string>`. You can also narrow scope by specifying warnings be turned on when generated by specific `<filename>`s.

It is also possible to turn warnings on or off in your MATLAB code.

For example, to turn warnings off for deployed applications (specified using `isdeployed`) in your `startup.m`, you write:

```
if isdeployed
    warning off
end
```

To turn warnings on for deployed applications, you write:

```
if isdeployed
    warning on
end
```

-win32 Run in 32-Bit Mode

Use this option to build a 32-bit application on a 64-bit system *only* when the following are true:

- You have a 32-bit installation of MATLAB.
- You use the same MATLAB installation root (`matlabroot`) for both 32-bit and 64-bit versions of MATLAB.
- You are running from a Windows command line.

-Y License File

Use

```
-Y license.lic
```

to override the default license file with the specified argument.

libraryCompiler

Purpose

Build and package functions for use in external applications

Syntax

`libraryCompiler` [-win32] [[[-build] | [-project]]*`project_name`*]

Description

`libraryCompiler` opens the MATLAB shared library compiler for the creation of a new compiler project

`libraryCompiler project_name` opens the MATLAB shared library compiler app with the project preloaded.

`libraryCompiler -build project_name` runs the MATLAB shared library compiler to build the specified project. The installer is not generated.

`libraryCompiler -package project_name` runs the MATLAB shared library compiler to build and package the specified project. The installer is generated.

`libraryCompiler -win32` instructs the compiler to build a 32-bit application on a 64-bit system when you use the same MATLAB installation root (*`matlabroot`*) for both 32-bit and 64-bit versions of MATLAB.

Input Arguments

project_name - name of the project to be compiled

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

Purpose

Tests and diagnoses MATLAB Production Server instance for problems.

Syntax

`mps-check` [--timeout *`seconds`*] *`host`*:*`port`*

Description

`mps-check` sends a request to a MATLAB Production Server instance and receives a status report that is used to identify issues that cause the product to run less than optimally.

Information reported by `mps-check` to `stdout` include:

- Status of the server instance
- Port the HTTP interface is listening on
- Deployed archives for a server instance

Before using `mps-check`, you must deploy *`mcrroot`*`/bin/`*`arch`*`/mps_check.ctf` to the server instance.

- *`mcrroot`* is the path to the MCR's installation folder.
- *`arch`* is standard abbreviation for the system's operating system and hardware architecture.

Input Arguments

- `--timeout` *`seconds`* — The time, in seconds, to wait for a response from the server before timing out. The default is two minutes.
- *`host`* — The host name of the machine running the server instance.
- *`port`* — The port number on which the server instance listens for requests.

DefinitionsServer Instance

An instance of the MATLAB Production Server. The files contained in the folder created by `mps-new`, defined by *`path`*`/`, comprise one configuration of the MATLAB Production Server product.

Examples

Display diagnostic information for the server instance running on port 9910 of the local computer.

```
mps-check localhost:9910
```

```
Connecting to localhost:9910
Connected
Sending HTTP request
HTTP request sent
HTTP response received
MPS status check completed successfully
```

mps-license-reset

Purpose

Forces server to immediately attempt license checkout

Syntax

```
mps-license-reset [-C path/]server_name
```

Description

`mps-license-reset [-C` *`path/`*`]`*`server_name`* triggers the server to checkout a license immediately, regardless of the current license status. License keys that are currently checked out are checked in first.

Tips

- Run this command at your operating system prompt.

Input Arguments

-C *path/*

Specify a path to the server instance. If this option is omitted, the current working folder and its parents are searched to find the server instance.

server_name

Server checking out license

DefinitionsServer Instance

An instance of the MATLAB Production Server. The files contained in the folder created by `mps-new`, defined by *`path/`*, comprise one configuration of the MATLAB Production Server product.

Examples

Create a new server instance and display the status of each folder in the file hierarchy, as the server instance is created:

```
mps-license-reset -C /tmp/server_2
```

See Also

`mps-status` |

Related Examples

- "Forcing a License Checkout Using mps-license-reset" on page 5-10

Concepts

- “License Management for MATLAB® Production Server™” on page 5-9

Purpose

Create server instance

Syntax

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

Description

`mps-new [path/]server_name [-v]` makes a new folder at *`path`* and populates it with the default folder hierarchy for a .

Each server instance can be configured, started, monitored, and stopped independently.

Tips

- Before creating a server instance, ensure that no file or folder with the specified *`path`* currently exists on your system.
- After issuing `mps-new`, you must issue `mps-start` to start the server instance.
- Run this command at your operating system prompt.

Input Arguments

`path/`

Path to server instance.

`server_name`

Name of the server to be created.

If you are creating a server instance in the current working folder, you do not need to specify a full path. Only specify the server name.

-v

Displays status of each folder in the file hierarchy, created to form a server instance

DefinitionsServer Instance

An instance of the MATLAB Production Server. The files contained in the folder created by `mps-new`, defined by *`path/`*, comprise one configuration of the MATLAB Production Server product.

Examples

Create a new server instance and display the status of each folder in the file hierarchy, as the server instance is created:

```
mps-new /tmp/server_1 -v
```

Example Output

```
server_1/.mps-version...ok
server_1/config/...ok
server_1/config/main_config...ok
server_1/endpoint/...ok
server_1/auto_deploy/...ok
server_1/.mps-socket/...ok
server_1/log/...ok
server_1/pid/...ok
```

See Also

`mps-status` **|** `mps-start` **|**

Related Examples

- "Server Creation" on page 5-11

Concepts

- "Server Overview" on page 5-2

Purpose

Stop and start server instance

Syntax

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

Description

`mps-restart [-C path/]server_name [-f]` stops a server instance, then restarts the same server instance. Issuing `mps-restart` is equivalent to issuing the `mps-stop` and `mps-start` commands in succession.

Tips

- After issuing `mps-restart`, issue the `mps-status` command to verify the server instance has started.
- If you are restarting a server instance in the current working folder, you do not need to specify a full path. Only specify the server name.
- Run this command at your operating system prompt.

Input Arguments

-C *path/*

Specify a path to the server instance. If this option is omitted, the current working folder and its parents are searched to find the server instance. If you are restarting a server instance in the current working folder, you do not need to specify a full path. Only specify the server name.

server_name

Name of the server to be restarted.

-f

Force success even if the server instance is stopped. Restarting a stopped instance returns an error.

Definitions

Server Instance

An instance of the MATLAB Production Server. The files contained in the folder created by `mps-new` comprise a single configuration of the MATLAB Production Server product.

Examples

Restart a server instance named `server_1`, located in folder `tmp`. Force successful completion of `mps-restart`.

```
mps-restart -f -C /tmp/server_1
```

See Also

`mps-start` | `mps-stop` | `mps-status` |

Purpose

Sets up server environment

Syntax

```
mps-setup [mcrroot]
```

Description

`mps-setup [mcrroot]` sets location of MATLAB Compiler Runtime (MCR) and other start-up options.

The `mps-setup` command sets the default path to the MATLAB Compiler Runtime (MCR) for all server instances you create with the product. This is equivalent to presetting the `--mcr-root` option in each server's `main_config` configuration file.

If a default value already exists in *`server_name`*`/config/mcrroot`, it is updated with the value specified when you run the command line wizard.

Tips

- Run `mps-setup` from the script folder. Alternatively, add the `script` folder to your system `PATH` environment variable to run `mps-setup` from any folder on your system.
- Run `mps-setup` without arguments and it will search your system for MCR instances you may want to use with MATLAB Production Server.
- Run `mps-setup` by passing the path to the MATLAB Compiler Runtime (MCR) as an argument. This method is ideal for non-interactive (silent) installations.

Input Arguments

mcrroot

Specify a path to the MATLAB Compiler Runtime if running `mps-setup` in non-interactive, or silent, mode.

Definitions

Server Instance

An instance of the MATLAB Production Server. The files contained in the folder created by `mps-new`, defined by *`path`*`/`, comprise one configuration of the MATLAB Production Server product.

Examples

Run `mps-setup` non-interactively, by passing in a path to the MATLAB Compiler Runtime (MCR) instance that you want MATLAB Production Server to use.

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

`mcrver` is the version of the MCR to use.

See Also

`mps-start` | `mps-new` | `mps-status` |

Concepts

- “Run mps-setup to Set Location of MATLAB Compiler Runtime (MCR)” on page 5-6

Purpose

Starts server instance

Syntax

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

Description

`mps-start [-C path/]server_name [-f]` starts a server instance

Tips

- After issuing `mps-start`, issue the `mps-status` command to verify the server instance has `STARTED`.
- If you are starting a server instance in the current working folder, you do not need to specify a full path. Only specify the server name.
- Run this command at your operating system prompt.

Input Arguments

-C *path/*

Specify a path to the server instance. If this option is omitted, the current working folder and its parents are searched to find the server instance.

server_name

Name of the server to be started.

-f

Force success even if the server instance is currently running. Starting a running server instance is considered an error.

Definitions

Server Instance

An instance of the MATLAB Production Server. The files contained in the folder created by `mps-new` comprise a single configuration of the MATLAB Production Server product.

Examples

Start a server instance named `server_1`, located in folder `tmp`. Force successful completion of `mps-start`.

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

See Also

`mps-stop` | `mps-restart` | `mps-status` |

Related Examples

- “Start a Server” on page 5-28

Concepts

- “Server Startup” on page 5-27
- “Server Overview” on page 5-2

Purpose

Displays status of server instance

Syntax

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

Description

`mps-status [-C path/]server_name` displays the status of the server (`STARTED`, `STOPPED`), along with a full path to the server instance.

Tips

- If you are creating a server instance in the current working folder, you do not need to specify a full path. Only specify the server name.
- If the server is running, the status of the license associated with that server will also be displayed.
- Run this command at your operating system prompt.

Input Arguments

-C *path/*

Specify a path to the server instance. If this option is omitted, the current working folder and its parents are searched to find the server instance.

server_name

Server to be queried for status

Definitions

Server Instance

An instance of the MATLAB Production Server. The files contained in the folder created by `mps-new`, defined by *`path/`*, comprise one configuration of the MATLAB Production Server product.

Examples

Display status of server instance `server_1`, residing in `tmp` folder.

```
mps-status -C /tmp/server_1
```

Example Output

If server is running and running with a valid license:

```
'/tmp/server_1' STARTED
```

```
license checked out
```

If server is not running:

```
'/tmp/server_1' STOPPED
```

See Also

`mps-start` | `mps-stop` | `mps-restart` | `mps-which` |

Related Examples

- “Start a Server” on page 5-28

Concepts

- “Server Startup” on page 5-27
- “Server Overview” on page 5-2
- “License Server Status Information” on page 5-31

Purpose

Stop server instance

Syntax

```
mps-stop [-C path/]server_name [-f] [-v]
[--timeout hh:mm:ss]
```

Description

`mps-stop [-C path/]server_name [-f] [-v] [--timeout hh:mm:ss]` closes HTTP server socket and all open client connections immediately. All function requests that were executing when the command was issued are allowed to complete before the server shuts down.

Tips

- After issuing `mps-stop`, issue the `mps-status` command to verify the server instance has `STOPPED`.
- If you are stopping a server instance in the current working folder, you do not need to specify a full path. Only specify the server name.
- Run this command at your operating system prompt.
- Note that the timeout option (`--timeout` *hh*:*mm*:*ss*) is specified with two (2) dashes, not one dash.

Input Arguments

-C *path/*

Specify a path to the server instance. If this option is omitted, the current working folder and its parents are searched to find the server instance.

server_name

Name of the server to be stopped.

-f

Force success even if the server instance is not currently stopped. Stopping a stopped instance is considered an error.

-v

Displays system messages relating to termination of server instance.

--timeout *hh:mm:ss*

Set a limit on how long `mps-stop` will run before returning either success or failure. For example, specifying `--timeout 00:02:00` indicates that `mps-stop` should exit with an error status if the server takes longer than two (2) minutes to shut down. The instance continues to attempt to terminate even if `mps-stop` times out. If this option is not specified, the default behavior is to wait as long as necessary (infinity) for the instance to stop.

Definitions

Server Instance

An instance of the MATLAB Production Server. The files contained in the folder created by `mps-new`, defined by *`path`*`/`, comprise one configuration of the MATLAB Production Server product.

Examples

Stop server instance `server_1`, located in `tmp` folder. Force successful completion of `mps-stop`. Timeout with an error status if `mps-stop` takes longer than three (3) minutes to complete.

In this example, the verbose (`-v`) option is specified, which produces an output status message.

```
mps-stop -f -v -C /tmp/server_1 --timeout 00:03:00
```

Example Output

```
waiting for stop... (timeout = 00:03:00)
```

See Also

`mps-start` | `mps-restart` | `mps-new` | `mps-status` |

mps-support-info

Purpose

Displays licensing and configuration information of a MATLAB Production Server instance

Syntax

`mps-support-info` [-C [[*`instance_path`*] | [*`server_name`*]]]

Description

`mps-support-info` displays licensing and configuration information of a MATLAB Production Server instance.

Input Arguments

- `-C` *`instance_path`* — The path to where the server instance is installed.
- `-C` *`server_name`* — The name of the server instance to locate in the current folder.

Definitions

Server Instance

An instance of the MATLAB Production Server. The files contained in the folder created by `mps-new`, defined by *`path`*`/`, comprise one configuration of the MATLAB Production Server product.

Examples

Display licensing and configuration information of server instance `fred`, residing in `/` folder.

```
mps-status -C /fred

Instance Version:           1.0
License Number:             UNKNOWN -- MPS stopped
MPS Version:                UNKNOWN -- MPS stopped
Available License Number:   857812
Client Version:             1.0.1 R2013a
Operating System:           Microsoft Windows 7 Enterprise Edition (bu
Number of CPU cores:        8
CPU Info:                   Intel(R) Xeon(R) CPU           W3550  @ 3
Memory:                     11.9915 GB ( 1.2574e+007 KB )
```

mps-which

Purpose

Display path to server instance that is currently using the configured port.

Syntax

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

Description

`mps-which [-C path/]server_name` is useful when running multiple server instances on the same machine. If you accidently leaves a server instance running and try to start another which is configured to use the same port number, the latter server instance will fail to start, displaying an `address-in-use` error. `mps-which` can be used to identify which server instance is using the port.

Tips

- If you are creating a server instance in the current working folder, you do not need to specify a full path. Only specify the server name.
- Run this command at your operating system prompt.

Input Arguments

-C *path/*

Specify a path to the server instance. If this option is omitted, the current working folder and its parents are searched to find the server instance.

server_name

Server to be queried for path.

Definitions

Server Instance

An instance of the MATLAB Production Server. The files contained in the folder created by `mps-new`, defined by *`path/`*, comprise one configuration of the MATLAB Production Server product.

Examples

`server_1` and `server_2`, both residing in folder `tmp`, are configured to use to same port, defined by `--http` in the `main_config` configuration files. However, the port can only be allocated to one server.

Run `mps-which` for both servers:

```
mps-which -C /tmp/server_1
```

```
mps-which -C /tmp/server_2
```

Example Output

In both cases, the server that has allocated the configured port displays (server_1):

```
/tmp/server_1
```

See Also

mps-status |

Data Conversion Rules

Conversion of Java Types to MATLAB Types

Value Passed to Java Method is:	Input type Received by MATLAB is:	Dimension of Data in MATLAB is:
java.lang.Byte, byte	int8	{1,1}
byte[] *data*		{1, *data.length*}
java.lang.Short short	int16	{1,1}
short[] *data*		{1, *data.length*}
java.lang.Integer, int	int32	{1,1}
int[] *data*		{1, *data.length*}
java.lang.Long, long	int64	{1,1}
long[] *data*		{1, *data.length*}
java.lang.Float, float	single	{1,1}
float[] *data*		{1, *data.length*}
java.lang.Double, double	double	{1,1}
double[] *data*		{1, *data.length*}
java.lang.Boolean, boolean	logical	{1,1}
boolean[] *data*		{1, *data.length*}
java.lang.Character, char	char	{1,1}
char[] *data*		{1, *data.length*}
java.lang.String *data*		{1, *data.length*()}

Value Passed to Java Method is:	Input type Received by MATLAB is:	Dimension of Data in MATLAB is:
`java.lang.String[]` *`data`*	`cell`	{1, *`data.length`*}
`java.lang.Object[]` *`data`*		{1, *`data.length`*}
`T[]` *`data`* [1]	MATLAB type for `T` [1]	{ *`data.length`*, *`dimensions`*(`T[0]`) }, if `T` is an array
		{ 1, *`data.length`*}, if `T` is not an array

[1] Where `T` represents any supported MATLAB type. If T is an array type, then all elements of data must have exactly the same length

1.

Conversion of MATLAB Types to Java Types

When MATLAB Returns:	Dimension of Data in MATLAB is:	MATLAB Data Converts To Java Type:
int8, uint8	{1,1}	byte, java.lang.Byte
	{1,*n*} , {*n*,1}	byte[*n*], java.lang.Byte[*n*]
	{*m*,*n*,*p*,...}	byte[*m*][*n*][*p*]... , java.lang.Byte[*m*][*n*][*p*]...
int16, uint16	{1,1}	short, java.lang.Short
	{1,*n*} , {*n*,1}	short[*n*], java.lang.Short[*n*]
	{*m*,*n*,*p*,...}	short[*m*][*n*][*p*]... , java.lang.Short[*m*][*n*][*p*]...
int32, uint32	{1,1}	int, java.lang.Integer
	{1,*n*} , {*n*,1}	int[*n*], java.lang.Integer[*n*]
	{*m*,*n*,*p*,...}	int[*m*][*n*][*p*]... , java.lang.Integer[*m*][*n*][*p*]...
int64, uint64	{1,1}	long, java.lang.Long
	{1,*n*} , {*n*,1}	long[*n*], java.lang.Long[*n*]
	{*m*,*n*,*p*,...}	long[*m*][*n*][*p*]... , java.lang.Long[*m*][*n*][*p*]...
single	{1,1}	float, java.lang.Float
	{1,*n*} , {*n*,1}	float[*n*], java.lang.Float[*n*]
	{*m*,*n*,*p*,...}	float[*m*][*n*][*p*]... , java.lang.Float[*m*][*n*][*p*]...

When MATLAB Returns:	Dimension of Data in MATLAB is:	MATLAB Data Converts To Java Type:
double	{1,1}	double, java.lang.Double
	{1,*n*} , {*n*,1}	double[*n*], java.lang.Double[*n*]
	{*m*,*n*,*p*,...}	double[*m*][*n*][*p*]... , java.lang.Double[*m*][*n*][*p*]...
logical	{1,1}	boolean, java.lang.Boolean
	{1,*n*} , {*n*,1}	boolean[*n*], java.lang.Boolean[*n*]
	{*m*,*n*,*p*,...}	boolean[*m*][*n*][*p*]... , java.lang.Boolean[*m*][*n*][*p*]...
char	{1,1}	char, java.lang.Character
	{1,*n*} , {*n*,1}	java.lang.String
	{*m*,*n*,*p*,...}	char[*m*][*n*][*p*]... , java.lang.Character[*m*][*n*][*p*]...
cell (containing only strings)	{1,1}	java.lang.String
	{1,*n*} , {*n*,1}	java.lang.String[*n*]
	{*m*,*n*,*p*,...}	java.lang.String[*m*][*n*][*p*]...
cell (containing multiple types)	{1,1}	java.lang.Object
	{1,*n*} , {*n*,1}	java.lang.Object[*n*]
	{*m*,*n*,*p*,...}	java.lang.Object[*m*][*n*][*p*]...

Conversion Between MATLAB Types and C# Types

This MATLAB type....	Is equivalent to this C# type....
uint8	byte
int8	sbyte
uint16	ushort
int16	short
uint32	uint
int32	int
uint64	ulong
int64	long
single	float
double	double
logical	bool
char	System.String, char
cell (strings only)	Array of System.String
cell (heterogeneous data types)	Array of System.Object
struct	A .NET struct or class with public fields or public properties

Note Multidimensional arrays of above C# types are supported. Jagged arrays are not supported.

MATLAB Production Server .NET Client API Classes and Methods

MATLABException

About MATLABException

Use `MATLABException` to handle MATLAB exceptions thrown by .NET interfaces

Errors are thrown during invocation of MATLAB function associated with a MATLAB Production Server request initiated by `MWHttpClient`.

MATLAB makes the following information available in case of an error:

- MATLAB stack trace
- Error ID
- Error message

Derived from `Exception`

Members

Constructor

```
public MATLABException(
       string, message
       string, identifier
       IList<MATLABStackFrame> stackList
);
```

Creates an instance of `MATLABException` using MATLAB error message, error identifier, and a list of `MATLABStackFrame`, representing MATLAB stack trace associated with a MATLAB error.

Constructor Parameters

string, message

Error message from MATLAB

string, identifier

Error identifier used in MATLAB

IList<MATLABStackFrame> stackList

List of `MATLABStackFrame` representing MATLAB stack trace. An unmodifiable copy of this list is made

Public Instance Properties

MATLABStackTrace

Returns list of `MATLABStackFrame`

Gets MATLAB stack with `0` or more `MATLABStackFrame`.

Each stack frame provides information about MATLAB file, function name, and line number. The output list of `MATLABStackFrame` is unmodifiable.

Message

Returns detailed MATLAB message corresponding to an error

MATLABIdentifier

Returns identifier used when error was thrown in MATLAB

MATLABStackTraceString

Returns string from stack trace

Public Instance Methods

None

Requirements

Namespace

`com.mathworks.mps.client`

Assembly

`MathWorks.MATLAB.ProductionServer.Client.dll`

See Also

`MATLABStackFrame`

MATLABStackFrame

About MATLABStackFrame

Use `MATLABStackFrame` to return an element in MATLAB stack trace obtained using `MATLABException`.

`MATLABStackFrame` contains:

- Name of MATLAB file
- Name of MATLAB function in MATLAB file
- Line number in MATLAB file

Members

Constructor

```
public MATLABStackFrame(
   string, file
   string , name
   int line
);
```

Construct `MATLABStackFrame` using file name, function name, and line number

Constructor Parameters

***string*, file**

Name of the file

***string*, name**

Name of function in the file

int **line**

Line number in MATLAB file

Public Instance Properties

File

Returns complete path to MATLAB file

Name

Returns name of a MATLAB function in a MATLAB file

For a MATLAB file with only one function, `Name` is equivalent to the MATLAB file name, without the extension. The name will be different from the MATLAB file name if it is a sub function in a MATLAB file.

Line

Returns a line number in a MATLAB file

Public Instance Methods

ToString

```
public override string ToString()
```

Returns a string representation of an instance of `MATLABStackFrame`

Equals

```
public override bool Equals(object obj)
```

Returns `true` if two `MATLABStackFrame` instances have the same file name, function name, and line number

GetHashCode

```
public override int GetHashCode()
```

Returns hash value for an instance of `MATLABStackFrame`

Requirements

Namespace

`com.mathworks.mps.client`

Assembly

`MathWorks.MATLAB.ProductionServer.Client.dll`

See Also

`MATLABException`

MWClient

About MWClient

Interface of `MWHttpClient`, providing client-server communication for MATLAB Production Server.

Members

Public Instance Methods

CreateProxy

```
T CreateProxy<T>(Uri url);
```

Returns a proxy object that implements interface `T`.

Creates a proxy object reference to the generic CTF archive hosted by the server. The CTF archive is identified by a URL.

The methods in returned proxy object match the names of MATLAB functions in the CTF archive that the user wants to deploy, as well as inputs and outputs consistent with MATLAB function types and values.

When these methods are invoked, the proxy object:

1. Establishes a client-server connection
2. Sends MATLAB function inputs to the server
3. Receives the results

Parameter List

- `T` — Type of the returned object
- `url` — URL to the CTF archive, with the form of `http://`*`localhost`*`:`*`port_number`*`/`*`CTF_archive_name_without_extension`*

Close

```
void Close();
```

Closes connection with the server.

Requirements

Namespace

`com.mathworks.mps.client`

Assembly

`MathWorks.MATLAB.ProductionServer.Client.dll`

See Also

`MWHttpClient`

MWHttpClient

About MWHttpClient

Implements `MWClient` interface.

Establishes HTTP-based connection between MATLAB Production Server client and server. The client and server can be hosted on the same machine, or different machines with different platforms.

`MWHttpClient` allows the client to invoke MATLAB functions exported by a generic CTF archive hosted by the server. The CTF archive is made available to the client as a URL.

A server can host multiple CTF archives since each CTF has a unique URL.

In order to establish client-server communication, the following is required:

- URL to the CTF archive in the form: `http://`*`localhost`*`:`*`port_number`*`/`*`CTF_archive_name_without_extension`*
- Names of MATLAB functions exported by the CTF archive
- Information about the number of inputs and outputs for each MATLAB function and their types
- A user-written interface including:
  - Public methods with same names matching those of the MATLAB functions exported by the CTF. Methods must be consistent with MATLAB functions in terms of the numbers of inputs and outputs and their types
  - Each method in this interface should declare the exceptions:
    - `Mathworks.MPS.Client.MATLABException` — Represents MATLAB errors
    - `System.Net.WebException` — Represents any transport errors during client-server communication
  - There can be overloads of a method in the interface, depending on the MATLAB function that the method is representing

  - Interface name does not have to match the CTF archive name

Members

Constructor

```
public class MWHttpClient : MWClient
```

Creates an instance of `MWHttpClient`

Public Instance Methods

CreateProxy

```
T CreateProxy<T>(Uri url);
```

Returns a proxy object that implements interface `T`.

Creates a proxy object reference to the generic CTF archive hosted by the server. The CTF archive is identified by a URL.

The methods in returned proxy object match the names of MATLAB functions in the CTF archive that the user wants to deploy, as well as inputs and outputs consistent with MATLAB function types and values.

When these methods are invoked, the proxy object:

1. Establishes a client-server connection

2. Sends MATLAB function inputs to the server

3. Receives the results

Parameter List

- `T` — Type of the returned object
- `url` — URL to the CTF archive, with the form of `http://`*`localhost`*`:`*`port_number`*`/`*`CTF_archive_name_without_extension`*

Close

```
void Close();
```

Closes connection with the server.

Requirements

Namespace

`com.mathworks.mps.client`

Assembly

`MathWorks.MATLAB.ProductionServer.Client.dll`

See Also

`MWClient`

MWStructureListAttribute

About MWStructureListAttribute

`MWStructureListAttribute` provides .NET types, which are convertible to and from MATLAB structures.

`MWStructureList` is used when a variable of declared type `System.Object` (scalar or multi-dimensional) either refers to or contains another MATLAB-struct-convertible type (a user-defined .NET struct or class) at run time.

`MWStructureListAttribute` allows you to scope data conversion at field, property, method, or interface level.

Members

Constructor

```
public MWStructureListAttribute(
   params Type[] structTypes
);
```

Construct `MWStructureListAttribute` using an array of user-defined types (`structTypes`).

Requirements

Namespace

`com.mathworks.mps.client`

Assembly

`MathWorks.MATLAB.ProductionServer.Client.dll`

Index