MATLAB® Production Server™
User's Guide

R2012b

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™

Revision History

September 2012	Online only	New for Version 1.0 (Release R2012b)

Contents

5 Client Programming

Introducing MATLAB Production Server

Product Description

Run MATLAB® programs as a 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

Product Overview

In this section...
"How Does This Product Work?" on page 1-3
"Who Uses this Product?" on page 1-3

How Does This Product Work?

MATLAB Production Server run your MATLAB applications in a production environment by hosting the MATLAB Compiler Runtime (MCR), a standalone set of shared libraries that enables the execution of compiled MATLAB applications or components on computers that do not have MATLAB installed.

MATLAB Production Server works in conjunction with MATLAB Compiler to build CTF archives of MATLAB functions and deploying them in a scalable production server environment. It does this by:

- Building MATLAB function libraries, creating a deployable CTF archive using the Deployment Tool. The archive is generic because it can be accessed by various third-party clients in the environment you select. Currently, Java and Microsoft® .NET Framework are supported.
- Executing MATLAB functions, on demand, from instances of the MATLAB Compiler Runtime (MCR), managed on the server.

Who Uses this Product?

Tasks performed by users of MATLAB Production Server can be subdivided into three functional areas, or roles.

Depending on your installation, one user may perform one, two, or all three roles. Different roles assume different skill sets and expertise. See the following table for detailed descriptions of tasks performed by each type of user.

How Different Roles Work with MATLAB Production Server

This Type of User....	Uses MATLAB Production Server To....
MATLAB programmer	Deploys MATLAB functions to enterprise and Web production environments. Using the Deployment Tool in MATLAB, the MATLAB programmer creates a CTF archive from MATLAB code. This archive is hosted by server instances in the MATLAB Production Server product.
Server administrator	Manages CTF archives in the `auto-deploy` repositories on the server, in a production environment. Using commands and diagnostic tools, the administrator maintains the server environment in order to host deployed archives and the MATLAB Compiler Runtime (MCR) for clients.
Application developer	Develops client interfaces in Java or C# (using Microsoft .NET Framework). The client interface is modeled on the MATLAB function that is hosted. The application developer integrates the client interface into larger enterprise and Web applications, as needed. Once an archive is hosted by MATLAB Production Server, any changes made to the compiled MATLAB code are immediately available on the server, and so through the client interface.

User Roles and Tasks

MATLAB Programmer

Role	Knowledge Base	Tasks Can Include:
👤 **MATLAB programmer**	• MATLAB expert • No IT experience needed • No access to IT systems	• Installs MATLAB Compiler • Writes and tests MATLAB code • Designs application, with Java or .NET developer • Creates a deployable CTF archive using the Deployment Tool • Hands off the CTF archive to the Java or .NET developer

Java or .NET Developer

Role	Knowledge Base	Tasks Can Include:
Java developer **.NET developer**	• No MATLAB experience needed • Moderate IT Experience preferable • Proficient in client coding • Minimal access to IT systems	• Installs client SDK or IDE • Creates client interface • Designs application, with MATLAB programmer • Writes application code • Installs and tests application • Installs and tests application with MATLAB Production Server Software

Server Administrator

Role	Knowledge Base	Tasks Can Include:
Server administrator	• No MATLAB experience needed • Access to IT Systems • IT expert	• Installs the MATLAB Production Server and MATLAB Compiler Runtime (MCR) • Starts and shuts down MATLAB Production Server • Verifies MATLAB Production Server is running properly • Maintains the server, troubleshooting problems by examining log data.

Getting Started With MATLAB Production Server

Deploy MATLAB Code with MATLAB Production Server

In this section...

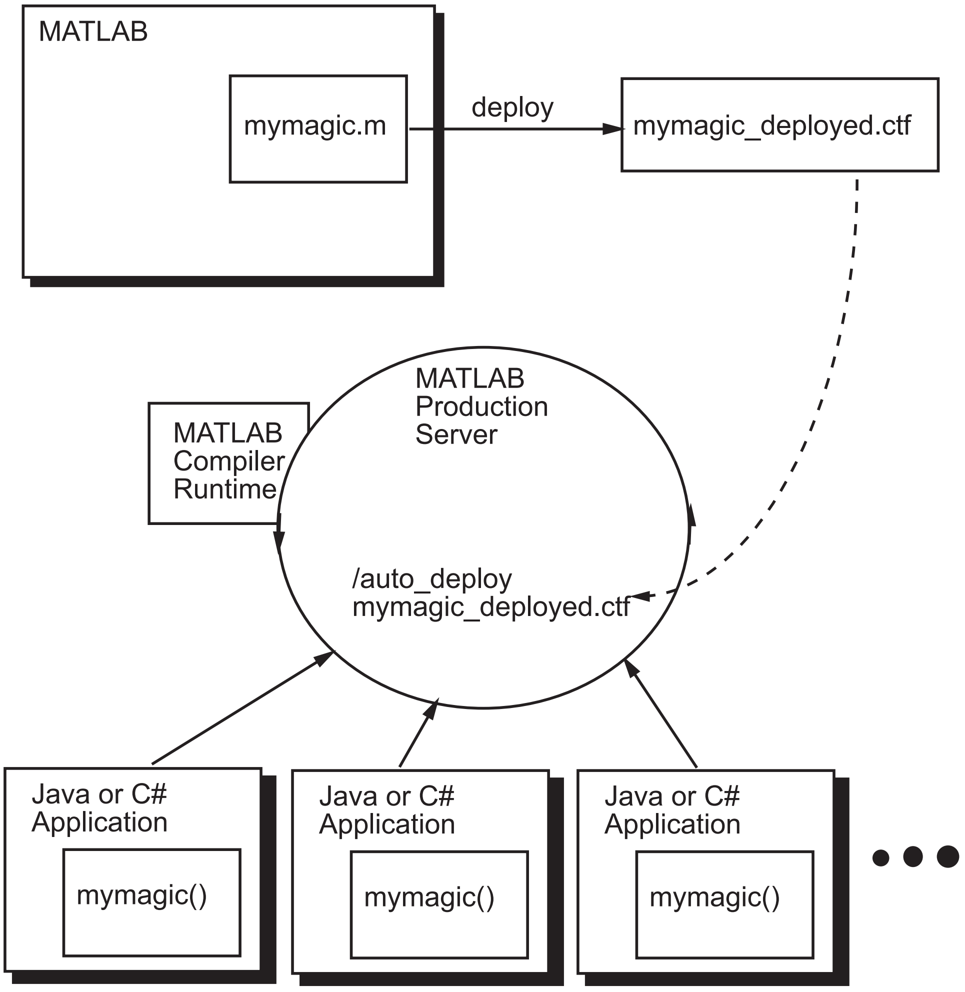
Introduction to the Workflow

This tutorial shows how to deploy MATLAB code using the MATLAB Production Server. The tutorial is made up of three smaller examples that illustrate the three parts of the process:

- Creating a deployable archive using MATLAB Compiler.
- Creating, configuring, and starting an instance of the MATLAB Production Server.
- Writing a Java or C# application, called a *client* in this documentation, that uses the deployed MATLAB code via the server.

To illustrate the workflow, this tutorial walks through all these required tasks. In a production environment, each task might be performed by separate individuals. For example, a MATLAB programmer might create the MATLAB function and deploy it, a system administrator might install and manage the server instance, and Java or C# programmers might create the client applications.

The following figure illustrates this workflow using the tutorial example, `mymagic.m`. In the figure, note how multiple client applications can call the same deployed function through a single server instance.



Create a Deployable CTF Archive

This example shows how to deploy a MATLAB function so that it can be used with the MATLAB Production Server. This task is typically performed by a MATLAB programmer. This part of the tutorial assumes you have MATLAB and the MATLAB Compiler installed on your system.

Write the MATLAB code you want to deploy

For this tutorial, the MATLAB function to be deployed returns a magic square of the size you specify. (A magic square is a matrix of integers arranged so that their sum is the same when added vertically, horizontally, or diagonally.) To work along with this tutorial, copy the following code into a file named `mymagic.m`, using the MATLAB editor, and save it in a folder of your choice.

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

Add the folder to the MATLAB path, so MATLAB can find it. To add a folder to the MATLAB path, double-click it in the **Current Folder** browser in MATLAB.

When you call `mymagic`, specifying a size, the function returns a magic square of that dimension.

```
mymagic(4)
```

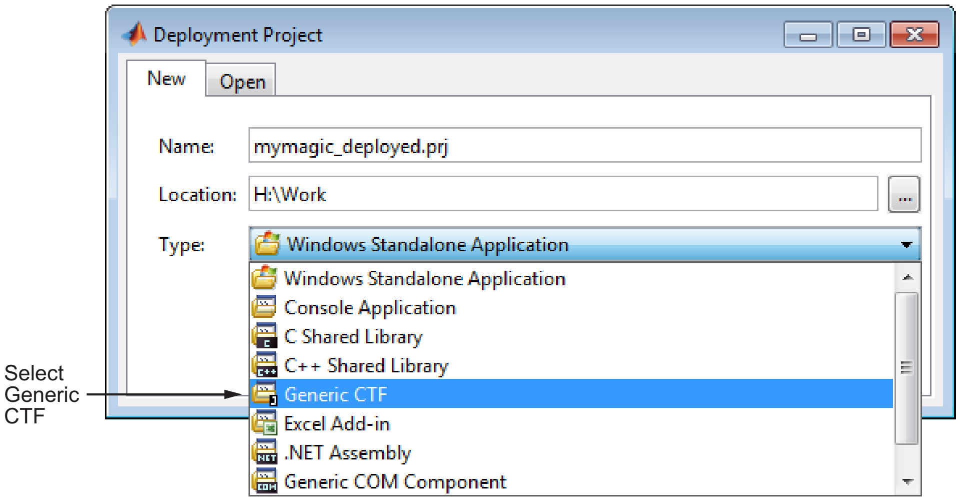
```
ans =

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

Deploy the function

At the MATLAB command prompt, type `deploytool` to launch the Deployment tool. In the Deployment Project dialog box, fill-in the required information, detailed in the following table, and click **OK**. The Compiler creates the deployment project file in the folder you specified.

Field	Value
Name	Specify the name you want to give your project. The example names the project `mymagic_deployed`
Location	Specify the folder where you want to store the project. The example uses the folder `H:\Work`.
Type	Specify the type of deployed application you want to create. To deploy using MATLAB Production Server, you must select **Generic CTF**

The following figure illustrates the settings for the example.



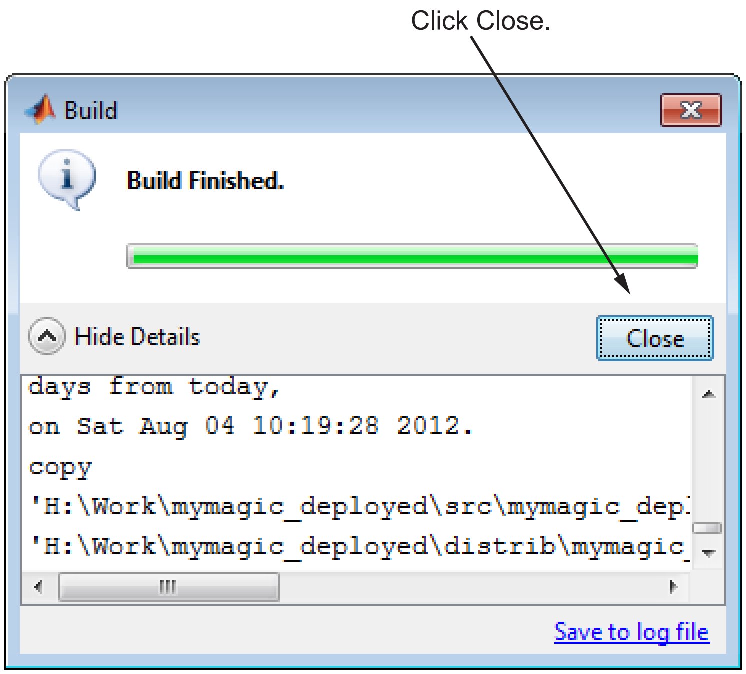
Name: mymagic_deployed.prj

Location: H:\Work

Type: Windows Standalone Application

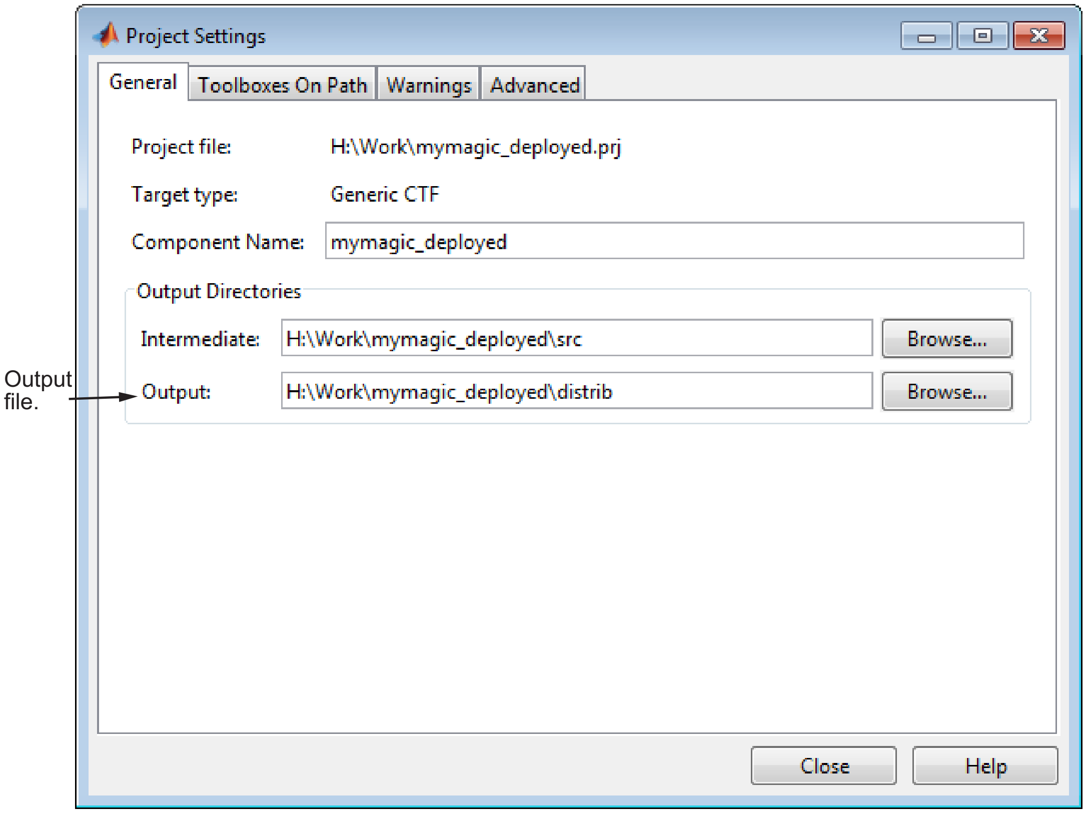
After creating your deployment project, you must add files to the project. On the **Build** tab, click **Add files**. In the Add Files dialog box, select the file or files you want to deploy and click **Open**. For this example, select `mymagic.m`.

After adding files, you can build the generic CTF archive. Click the Build button (). The Deployment tool displays a status dialog box during the build process. When the build finishes, click **Close** to dismiss the dialog box.



```
days from today,
on Sat Aug 04 10:19:28 2012.
copy
'H:\Work\mymagic_deployed\src\mymagic_dep
'H:\Work\mymagic_deployed\distrib\mymagic_
```

The Deployment tool creates the CTF file in the `distrib` folder of your deployment project. To view details about your deployment project, click the Action icon ( ) on the toolbar, and then click **Settings**.



Project file: H:\Work\mymagic_deployed.prj

Target type: Generic CTF

Component Name: mymagic_deployed

Intermediate: H:\Work\mymagic_deployed\src

Output: H:\Work\mymagic_deployed\distrib

For information about how to share your CTF archive using the MATLAB Production Server, see “Share the CTF Archive on the Server Instance” on page 2-10

Start a Server Instance

This example shows how to install, configure, and start an instance of MATLAB Production Server. This is called a *server instance* because you can have multiple servers running at the same time. This task is typically performed by a system administrator.

Install MATLAB Production Server

Run the installer and specify the folder into which you want to install MATLAB Production Server. During installation, on the Installation Type dialog box, choose to perform a custom installation because you must select the license manager for installation in the product list. This example uses the default installation folder, `C:\Program Files\MATLAB\MATLAB Production Server`. For more information about the installation process, see "Product Installation and Licensing" on page 4-5.

Note To run server commands from any folder on your computer, add the `MPS_root\script` folder to your system `PATH` environment variable, where `MPS_root` represents your MATLAB Production Server installation folder.

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. For information about obtaining the MCR, visit the MATLAB Compiler Runtime page.

Create a Server Instance

To create the server instance, move to the folder where you want to create your server and use the `mps-new` command. You specify the name of your server as an argument.

For this example, navigate to the `C:\tmp` folder and enter the following command. By specifying the `-v` option, the example displays the results of the command as each folder in the hierarchy is built.

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

```
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 “What is a Server?” on page 4-2

Configure the Server

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. For more information about configuration options, see “Configuration File Customization” on page 4-14. At a minimum, you must use the file to specify the location of the MCR you want to use with the server.

To configure a server, open your server’s configuration file, `main_config`, using a text editor of your choice. This file resides in `MPS_server_root\config`, where `MPS_server_root` represents the path to your server. For this example, the file is in `C:\tmp\prod_server_1\config`.

To specify the location of your MCR, set the value of the `--mcr-root` option in the file to the full path of your MCR. You *must* include the version number of the MCR (v*nnn*) in the path.

For this example, set the value as follow:

```
--mcr-root C:\Program Files\MATLAB\MATLAB Compiler Runtime\v80
```

You can also use the `mps-setup` command to specify the MCR location in `main_config`.

Save your changes to the`main_config` file and exit your text editor.

Start the Server

To start the server, use the `mps-start` command. You can either move into the server folder to execute the command, or use the `-C` option to specify the path to the server on the command line.

For example, to start `prod_server_1`, enter this command:

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

To ensure the server has started, use the `mps-status` command.

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

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

Share the CTF Archive on the Server Instance

To make your CTF archive available using MATLAB Production Server, you must copy the CTF file into the `auto_deploy` folder in your server instance. You can add a CTF into the `auto_deploy` folder of a running server—the server monitors this folder dynamically and processes the CTF files that are added to the `auto_deploy` folder.

For this example, copy the CTF file from the deployment project `H:\Work\mymagic_deployed\distrib\` folder into the server's `auto_deploy` folder, `C:\tmp\prod_server_1\auto_deploy`.

Create a Java Application That Calls the Deployed Function

This example shows how you can call a deployed MATLAB function from a Java application using MATLAB Production Server In your Java code, you must:

- Define a Java interface that represents the MATLAB function.
- Instantiate an object of the client class `MWHttpClient` and setup communication with the server through a proxy.
- Call the deployed function in your Java code.

This task is typically performed by Java application programmer. This part of the tutorial assumes you have the Java SDK installed on your computer.

Design a Java interface

Design a Java interface that represents the deployed MATLAB function. An interface declares the existence of a particular function that is implemented elsewhere.

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

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

might look like this:

```
interface MATLABMagic {
       double[][] mymagic(int size)
         throws MATLABException, IOException;
   }
```

When creating the interface, note the following:

- You can give the interface any valid Java name. The name doesn't have to be same as the name of CTF. This example specifies the name `MATLABMagic`
- You must give the method defined by this interface the same name as the deployed MATLAB function. For this example, that's `mymagic`.
- The Java method must support the same inputs and outputs supported by the MATLAB function, in both type and number. In MATLAB, you do not need to specify the data types of inputs and outputs. The MATLAB `mymagic` function accepts inputs of any numeric type and returns a matrix of type `double`. In Java, you must declare the data types of inputs and outputs. The Java `mymagic` method declares the input of type `int` and the output of type `double`, which are types supported by the MATLAB function. For more information about data type conversions and handling more complex MATLAB function signatures, see "Java Client" on page 5-4.
- The Java method `mymagic` must handle MATLAB exceptions and I/O exceptions, for any transport error during client-server communication.

Instantiate the Client Class and Create the Proxy Object

Instantiate a client object using the `MWHttpClient` constructor. This class establishes an HTTP connection between the application and the server instance.

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

After creating the object, call the 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 file as arguments:

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

For more information about the `createProxy` method, see the Javadoc included in the `MPS_root\client\java\doc` folder, where `MPS_root` is the name of your MATLAB Production Server installation folder.

Call the Deployed function in Java code

You call the deployed function in your Java application by calling the public method of the interface. In this example, the Java application calls `mymagic`, specifying the size of the magic square as an argument.

```
double[][] result = m.mymagic(siz);
```

Best practice is to free system resources after you are done calling the deployed function by calling the `close` method of the client object.

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

The following presents the full Java application to create a magic square, using MATLAB Production Server.

Java Class MPSClientExample

```
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 MATLABMagic {
    double[][] mymagic(int size) throws IOException, MATLABException;
}

public class MPSClientExample {

    public static void main(String[] args){

        if(args.length != 1){
            System.out.println("Usage: MPSClientExample size");
            System.exit(0);
        }
        int siz;
        siz = Integer.parseInt(args[0]);

        // Create a MWHttpClient instance
        MWClient client = new MWHttpClient();

        try{
            // Create the proxy object that represents magic.ctf
            MATLABMagic m = client.createProxy(new URL("http://localhost:9910/mymagic_deployed")
                                                        MATLABMagic.class );
            // The proxy object has mymagic as one of its public methods.
            //   Invocation of mymagic
            // results in a server request that sends magic square in response
            double[][] result = m.mymagic(siz);

            // 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{

            // Close the client
            client.close();
        }
    }

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

```
                System.out.println();
            }
        }
    }
```

Compile the Java application

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 "C:\Program Files\MATLAB\MATLAB Production Server\R2012b
\client\java\mps_client.jar" MPSClientExample.java
```

Run the Java application

After successfully compiling your Java application, run it using the `java` command or your IDE. For example, enter the following (on one line):

```
H:\Work>java -classpath .;"C:\Program Files\MATLAB\MATLAB Production Server\R2012b
\client\java\mps_client.jar" MPSClientExample 4
```

The application returns the magic square at the console:

```
16.0 2.0 3.0 13.0
5.0 11.0 10.0 8.0
9.0 7.0 6.0 12.0
4.0 14.0 15.0 1.0
```

Create a .NET Application That Calls the Deployed Function

This example shows how you can call a deployed MATLAB function from a .Net application using MATLAB Production Server In your .Net 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 .NET 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`, write a .NET interface named `Magic`.

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

```
 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 italicized line in the code below, the URL value (`"http://localhost:9910/magic"`) contains both CTF archive name (`magic`) 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/magic"));
                       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.Close();
            }
            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
```

MATLAB Code Deployment

Write MATLAB Code for Deployment

In this section...

Deployment Coding Guidelines

MATLAB coding guidelines are essentially the same for both the deployment products and MATLAB Production Server with the 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 3-2 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 less complicated and specific.

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®.
- Most data that is created in MATLAB, or from MATLAB applications, can be saved using MATLAB LOAD and SAVE commands and MAT files. See "Use MATLAB Data Files (MAT Files) in Compiled Applications" in the MATLAB Compiler documentation for more information and an example of saving state in a MAT file.
- 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.

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

In this section...

Prerequisites for Deployable Archive Creation

Before you create a deployable archive to host your MATLAB code with MATLAB Production Server, ensure you have:

- Installed the product and the MATLAB Compiler Runtime (MCR)
- Written MATLAB code that is compliant with deployable code guidelines. See "Write MATLAB Code for Deployment " on page 3-2 for details.

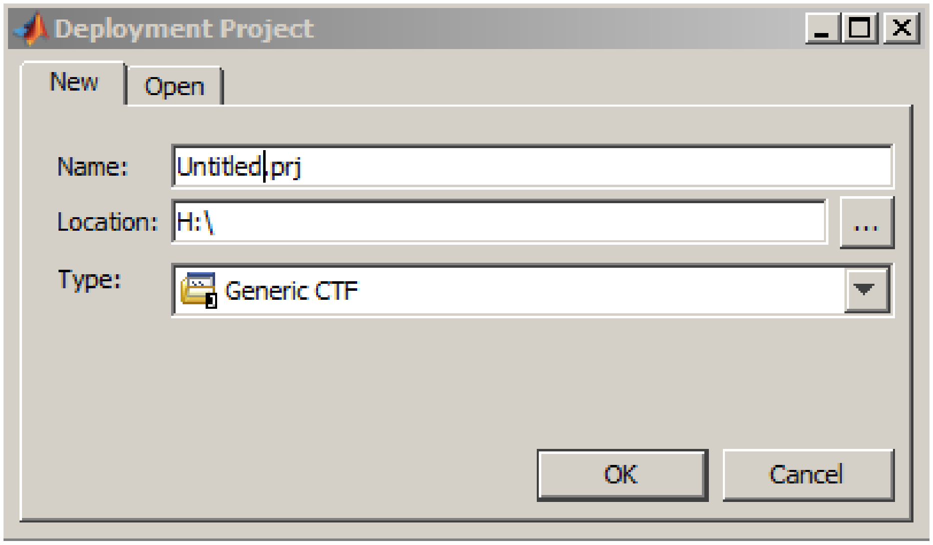
In order to share the deployable archive, a server must be created and started. See "Share the Deployable CTF Archive " on page 3-11 for a description of the complete workflow.

Build a Deployable CTF Archive

Do the following, to create a deployable CTF archive from MATLAB code.

1 Start MATLAB, if you have not done so already.

2 Type `deploytool` at the MATLAB command prompt, and press **Enter**. The Deployment Project dialog box opens.



Name: Untitled.prj

Location: H:\

Type: Generic CTF

The Deployment Project Dialog Box

3 Create a deployment project using the Deployment Project dialog box:

- **a** Type name of your project, in the **Name** field.
- **b** Enter the location of the project in the **Location** field.
- **c** Select **Generic CTF** as the target for the deployment project from the **Type** drop-down menu.

Note To deploy MATLAB functions with MATLAB Production Server, you must select **Generic CTF**.

- **d** Click **OK**.

Tip You can inspect the values in the Settings dialog before building your project. To do so, click the Action icon (⚙) on the toolbar, and then click **Settings**. Verify where your `src` and `distrib` folders will be created because you will need to reference these folders later.

---

4 On the **Build** tab:

   a Click **Add files** to open the Add Files dialog box.

   b Select the MATLAB file(s) you want to deploy and click **Open**.

5 When you complete your changes, click the Build button (). When the build finishes, click **Close** to dismiss the dialog box.

What Is a Deployable Archive?

A deployable archive is a compressed bundle of files created by the Deployment Tool. It is the same CTF archive created by MATLAB Compiler. The archive contains all the MATLAB-based content (MATLAB files, MEX-files, and so on) associated with the MATLAB function(s) being deployed. The generic archive is designed to be hosted by MATLAB Production Server instances and accessed by all supported clients. All MATLAB files are encrypted in the CTF archive using the Advanced Encryption Standard (AES) cryptosystem.

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 CTF archive you have built 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 your output folder (`distrib` folder) location.

See “Share the Deployable CTF Archive ” on page 3-11 for more information.

What Gets Built?

After you build your deployable CTF Archive with the Deployment Tool, you have the following files in the `src` and `distrib` subfolders of your project folder.

These Subfolders of the Project Folder...	Contain these files...
`src`	• *`project_name`*`.ctf` — Complete CTF archive (contains all source files — for internal use only) • Log files — for debugging
`distrib`	• *`project_name`*`.ctf` — Deployable CTF archive to be hosted by server in `auto_deploy` folder.

For More Information

If you want to...	See...
• Perform basic MATLAB Programmer tasks • Understand how the deployment products process your MATLAB functions • Understand how the deployment products work together • Explore guidelines about writing deployable MATLAB code	"Write Deployable MATLAB Code" in the *MATLAB Compiler User's Guide*.
Get help using the Deployment Tool	From the Deployment Tool, click the **Actions** icon (⚙) and select **Help**.

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 `distrib` subfolder of your Deployment Tool project folder. It will be named `*project_name*.ctf`. The locations of your project folders are defined in the Deployment Tool **Settings**.

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.

For More Information

If you want to learn more about...	See...
Installation and licensing for MATLAB Production Server	"Product Installation and Licensing" on page 4-5
What a server instance is and how it processes requests	"Server Overview" on page 4-2
Creating new server instances	"Server Creation" on page 4-10
Installing the MCR	"MATLAB Compiler Runtime (MCR) Installation" on page 4-13
MATLAB Production Server clients	"MATLAB® Production Server™ Client Overview" on page 5-2

Server Management

Server Overview

In this section...

What is a Server?

You can create any number of server instances using MATLAB Production Server. 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, or 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 parameter settings file (`main_config`) as well as its own set of diagnostic files (`log` files, Process Identification (`pid`) files, `endpoint` files).

In addition, each server has it's 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 its Work?

A server processes a transaction using these steps:

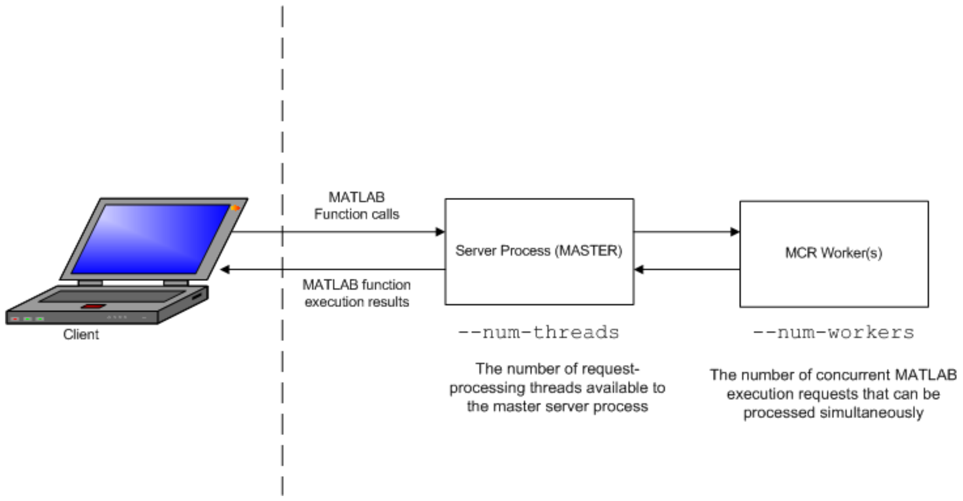
1. The client sends MATLAB function calls to the master server process (the main process on the server).

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 is passed back for processing by the client.


Client
MATLAB Function calls
MATLAB function execution results
Server Process (MASTER)
MCR Worker(s)
--num-threads
The number of request-processing threads available to the master server process
--num-workers
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 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 respectively for capacity and throughput.

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

  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 4-3).

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 or the number of request-processing threads available in the master server process. The threads in the pool do not execute MATLAB code directly; 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.

Product Installation and Licensing

In this section...

Install MATLAB Production Server

Installation Prerequisites

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.

Run the Installation Wizard

1. Insert the installation DVD into your computer. If the MathWorks® Installer does not automatically start, run `setup.exe`.

2. Follow the instructions in the Installation Wizard. For help completing the wizard, see the *MATLAB Installation Guide*. As you run the installation wizard, note the following:

   - If you do not already have the License Manager installed, it will be installed by default unless you specify otherwise, using the **Custom** installation option. The License Manager installed is the same program installed by default with an installation of MATLAB.

   - 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.

MATLAB Production Server version R2012b is only compatible with MCR version 8.0.

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 servers you create with the product.

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.

To run the command line wizard, do the following after first downloading and performing the "MATLAB Compiler Runtime (MCR) Installation" on page 4-13:

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\R2012b\script\mps-setup`.

3 Follow the instructions in the command line wizard.

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

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\v8.0"
```

Disable Windows Interactive Error Reporting (Optional)

If you run MATLAB Production Server on Windows, you may want to first disable Windows Interactive Error Reporting using the `DontShowUI` Windows Error Reporting (WER) setting.

If the system on which you are running MATLAB Production Server is not monitored frequently you may want to disable Windows Interactive Error Reporting 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.

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`.

License Management for MATLAB Production Server

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 running 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, if you specify this value for `--license`:

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

  the system will search 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 (before entering `hiberation` status). 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 (effectively entering `hiberation` status), before license server is polled, in order to verify and checkout a valid license. Polling will occur 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.

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, ensure you have completed "Product Installation and Licensing" on page 4-5.

Procedure

Before you can deploy your MATLAB code with MATLAB Production Server, 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 parameter settings file (`main_config`) as well as its own set of diagnostic files.

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

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

2. Enter the following command from the system prompt:

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

   where:

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

     If you are creating a server instance in the current folder, you do not need to specify a full path. Only specify the server name.

- *server_name* — is the name of the server instance and configuration you want to create.
- -v — enables verbose output, giving you information and status about each folder created in the server configuration.

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

Create a Server

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

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 "What is a Server?" on page 4-2

Note Before using a server, you must start it. See "Server Startup" on page 4-18.

For More Information

For information about....	See....
How to solve errors when creating a server	"Server Troubleshooting" on page 4-24
The `mps-new` command	`mps-new`
Product installation	"Product Installation and Licensing" on page 4-5

MATLAB Compiler Runtime (MCR) Installation

If you already have the MATLAB Compiler Runtime (MCR) installed, skip this step and “Configuration File Customization” on page 4-14.

Only MCR version 8.0 is compatible MATLAB Production Server version R2012b.

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.

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 need to specify to the server 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 the server configuration file.

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.

Specify the Installed MCR to Your 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 4-13 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. Examples of text editors include `vi`, Emacs, Wordpad, and Microsoft Visual Studio.

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

```
m C R r O O T u N s E T
```

6 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.

7 Save `main_config` and exit.

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
```

Keep the following in mind when editing `main_config`:

- 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 4-2 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 4-2 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 you would 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 4-13
Product installation	“Product Installation and Licensing” on page 4-5

Server Startup

In this section...
"Prerequisites" on page 4-18
"Procedure" on page 4-18
"Start a Server" on page 4-19
"For More Information" on page 4-19

Prerequisites

Before attempting to start a server, ensure you have:

- Installed the MATLAB Compiler Runtime (MCR)
- Created a server
- "Run mps-setup to Set Location of MATLAB Compiler Runtime (MCR)" on page 4-6

Procedure

To start a server, do the following:

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

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

   where:

   - `-C` *`path/`* is the path to the server instance and configuration you want to create for use with the MATLAB Production Server product. *`path`* should end with the server name.
   - *`server_name`* — is the name of the server instance and configuration 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 4-5
How to solve errors when starting or stopping a server	"Diagnose a Server Problem" on page 4-24 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	

Server Status Verification

In this section...

Prerequisite

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

Procedure

To verify the status of a server instance, do the following:

1. Open a system command prompt.

2. Enter the following command:

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

   where:

   - `-C path/` is the path to the server instance and configuration you want to create for use with the MATLAB Production Server product. *`path`* should end with the server name.
   - *`server_name`* — is the name of the server instance and configuration you want to start or stop.

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

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 `prod_server_1`'s status, from a location other than the server instance folder (`C:\tmp\prod_server_1`).

1 Open a system command prompt.

2 To ensure `prod_server_1` is running, enter the following command:

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

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

   ```
   \tmp\prod_server_1 STARTED
   ```

The output confirms `prod_server_1` is running. For more information on the `STOPPED` status and the `mps-stop` command, see `mps-stop` and `mps-restart`

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 has successfully been served a license and the license is assigned to the server whose status is being polled. This status indicates the server is operating with a valid license.
`Lost connection to the license server, in grace period`	The connection between server and license server has been lost. Licenses are currently being served for a limited grace period. License is assumed to be associated with the server.

This License Server Status Message....	**Means....**
`License server timeout, assuming the license reclaimed by the license server`	The server is no longer associated with a specific license or license server after the license server timeout has been reached or exceeded. Licenses are currently being served for a limited grace period. The license is assumed to no longer be assigned to the server and assumed to be reclaimed by the license server. For information about grace periods, see "Specify or Verify License Server Options in Server Configuration File" on page 4-8.
`License grace period has expired. HTTP requests temporarily disabled.`	The server is no longer associated with a specific license or license server and the grace period has expired. For information about grace periods, see "Specify or Verify License Server Options in Server Configuration File" on page 4-8.

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 the 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`. The active log files are created in the folder identified as`log-root` in `main_config`.

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`.

The archive log folder contains archived log with the auto-generated name of `main_`*`date_fileID`*`.log`, as well as a symbolic link to the latest archived log file, named `main_latest.log`.

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 4-26.

- Avoid placing `--log-root` and `--log-archive-root` on different physical file systems.
- Place log files on local, not 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 detail 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 4-18.

For More Information

For information about....	See....
The `mps-status` command	`mps-status`
Displaying which server has allocated a client port with the `mps-which` command	`mps-which`

Client Programming

MATLAB Production Server Client Overview

In this section...

What is a MATLAB Production Server Client?

MATLAB Production Server Clients are client applications written in a language supported by MATLAB Production Server (currently .NET/C# and Java) that call deployed functions hosted on a server.

Create a MATLAB Production Server Client

The following represents an overview of how to create a client with the MATLAB Production Server product.

1. Obtain the client run-time files, which are distributed with MATLAB Production Server and installed in *`$MPS_INSTALL`*`/client`.

2. Agree on the signatures of the MATLAB functions (with the MATLAB programmer) that comprise the services in the application. Each signature maps directly to a MATLAB function signature. See the prerequisites section in "Java Client" on page 5-4 and ".NET Client" on page 5-43 for specific requirements of each client.

3. Configure your system with the appropriate software for working with Java or .NET.

4. Write a Java or .NET interface and application program that creates a client (`MWClient`) by instantiating `MWHttpClient`.

   a. Create a dynamic proxy for communicating with the MATLAB Production Server-hosted service.

   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

In this section...

Java Client Coding Best Practices

When writing Java interfaces for invoking MATLAB code, keep the following in mind:

- 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

Do the following 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 (sometimes defined in separate GUIs, depending on your IDE).
3. 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 "Create a Deployable CTF Archive from MATLAB Code" on page 3-6.

   Your server's `main_config` file should point to where MATLAB or your MCR instance is installed (using `mcr-root`).
4. The server hosting your deployable CTF archive should 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(s) 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: • Stack trace • Error ID

Java Client Exceptions (Continued)

Exception	Reason(s) for Exception	Additional Information
		• 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

Where To Find the Example Code

All MATLAB and client code used to test and build client examples, can be found at `$MPS_INSTALL\examples`.

The example code listed in the documentation is not complete. Complete code is available in `$MPS_INSTALL\examples`, where `$MPS_INSTALL` is the location where you installed MATLAB Production Server.

Step 1: Write MATLAB Code

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

The code is available in `$MPS_INSTALL\examples\calculator`.

```
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 Deployment Tool

1 From MATLAB, start the Deployment Tool by entering `deploytool` at the MATLAB command prompt.

2 In the Deployment Project dialog box, create a project:

   a Enter `BondTools` in the **Name** field.

   b Select a location to store the project and enter it in the **Location** field.

   c In the **Type** drop-down, select **Generic CTF**.

   d Click **OK**.

The Deployment Tool creates your Generic CTF target project.

3 Add `pricecalc.m` to the deployment project:

   a On the **Build** tab, in the **Exported Functions** area, click **Add Files**.

   b In the Add Files dialog box, browse to `$MPS_INSTALL\examples\calculator`, select `pricecalc.m` and click **Open**.

4 In the Deployment Tool, click the Build icon (. When the build completes, you will find a file named `BondTools.ctf` in your project `distrib` 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 4-13 for more information.

2 Create a server using `mps-new`.

3 If you haven't 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 4-14 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 "Share the Deployable CTF Archive " on page 3-11 for complete details.

Step 4: Create the Java Client Code

Create a compatible client interface and method 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. The following example files can be found in `$MPS_INSTALL\examples\calculator`.

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`
`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 “MATLAB® Production Server™ Client Overview” on page 5-2, “Java Client” on page 5-4, and “Java Client Coding Best Practices” on page 5-4.

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

Declare Java Method Signatures Compatible with MATLAB Functions You Deploy. In order to work with the MATLAB functions you defined in "Step 1: Write MATLAB Code" on page 5-10, 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`. Note that there is a one-to-one mapping between the input arguments in both the MATLAB function and the Java interface, and that the interface specifies compatible type `double`. This compliance between the MATLAB and Java signatures demonstrates the guidelines listed in "Java Client Coding Best Practices" on page 5-4.

Instantiate MWClient, Create Proxy, and Specify Deployable CTF Archive. In the `ServerBondToolsFactory` class, you perform 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, specifying the port number (`9910`) and 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 5-13 when it is no longer needed.

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

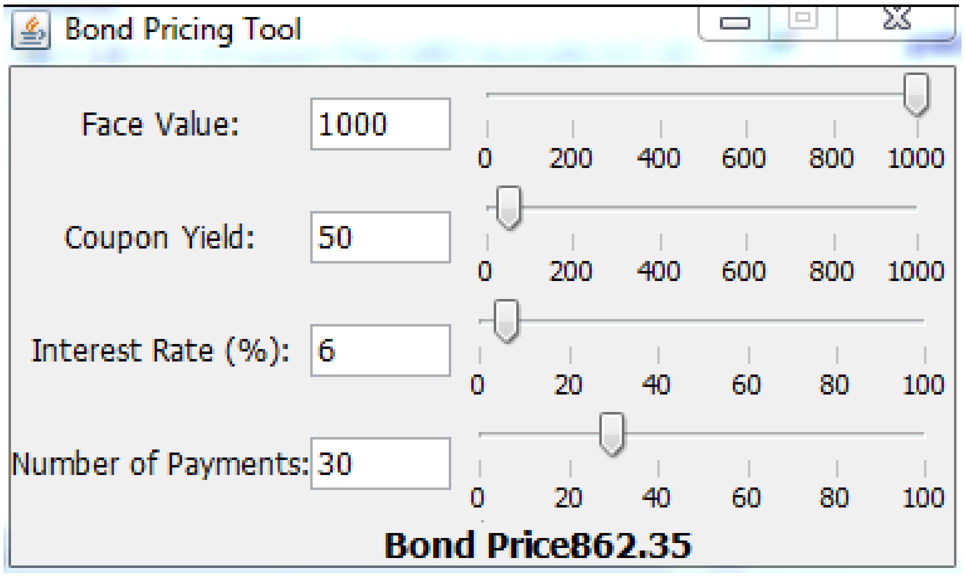
Step 5: Build the Client Code and Run the Example

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.

For more information, see "Java Client Prerequisites " on page 5-4.

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


Bond Pricing Tool

Face Value: 1000

Coupon Yield: 50

Interest Rate (%): 6

Number of Payments: 30

Bond Price862.35

Monte Carlo Simulation for Java Client

Monte Carlo simulations are useful for modeling systems whose inputs vary widely in range and scope. When working with such systems, it is often difficult to make accurate predictions due to the volatile nature of the data. In the financial industry, for example, the Monte Carlo method can be used to create realistic data sets used for trend analysis when working with fluctuating securities.

This example uses MATLAB Production Server to perform a Monte Carlo simulation on a number of stocks in a portfolio, yielding Value at Risk (*VaR*) and marginal value at risk (*mVaR*) at various confidence levels.

The values for *VaR* and *mVaR* are then plotted in a MATLAB figure.

This example shows how to create a Monte Carlo simulation that calculates *VaR* and *mVaR* concurrently using multiprocessing capabilities of the server and multithreading capabilities on the client.

Objectives

The Monte Carlo Simulation demonstrates the following:

- Invoking MATLAB functions concurrently with MATLAB Production Server
- Deploying MATLAB functions with multiple outputs
- Deploying a MATLAB function with graphical output

Where To Find the Example Code

All MATLAB and client code used to test and build client examples, can be found at `$MPS_INSTALL\examples`.

The example code listed in the documentation is not complete. Complete code is available in `$MPS_INSTALL\examples`, where `$MPS_INSTALL` is the location where you installed MATLAB Production Server.

Step 1: Write MATLAB Code

Before writing the client code, you write and deploy the Monte Carlo simulation code in MATLAB.

Modify the values in `mpsdemo_setup_mvar.m` as appropriate to the investment you want to model, or accept the default values in the program. Values include variables such as `numTimes`, `numSims`, and `time`. See the code comments for more information.

All files in this example can be found in `$MPS_INSTALL\examples\montecarlo\`.

This MATLAB code....	Provides this functionality....
`MPS_MVAR_demo_setup.m`	Prepares the Monte Carlo input data. Data is supplied in the associated MAT file (`mpsdemo_data_mvar.mat`).
`mpsdemoconfig.m`	Defines default configuration parameters used by the setup code (`MPS_MVAR_demo_setup.m`) such as number of tasks, number of simulations, and so on. One task is equivalent to one execution of `mpsdemo_task_mvar.m`.
`mpsdemo_setup_mvar.m`	Defines values related to the *mVar* calculations you want the Monte Carlo simulation to model (for example, number of simulations to perform, the number of times the simulations should be repeated, the stock price, proportionate weight of stock in portfolio and the confidence level at which we should calculate the value at risk). Calculates and returns level of difficulty used in the simulation.
`mpsdemo_helper_getDefaults.m`	Gets defaults from `mpsdemoconfig.m` to be used by `MPS_MVAR_demo_setup.m`.
`mpsdemo_helper_split_scalar.m`	Divides the simulation into concurrent tasks.
`mpsdemo_task_mvar.m`	Calculates *mVar* (marginal value at risk). One task is equivalent to one execution of `mpsdemo_task_mvar.m`. You set number of tasks by passing values to `mpsdemoconfig.m` (see following procedure).

This MATLAB code....	Provides this functionality....
`pTypeChecker.m`	Performs additional checks and calculations for `mpsdemo_task_mvar.m`.
`mpsdemo_plot_mvar.m`	Graphs the results of the Monte Carlo in a figure window.

Step 2: Create the Deployable CTF Archive that Runs the Simulation with the Deployment Tool

1. From MATLAB, start the Deployment Tool by entering `deploytool` using the MATLAB command prompt.
2. In the Deployment Project dialog box, create a project:
   a. Enter `BondTools` in the **Name** field.
   b. Select a location to store the project and enter it in the **Location** field.
   c. In the **Type** drop-down, select **Generic CTF**.
   d. Click **OK**.
   The Deployment Tool creates your Generic CTF target project.
3. Add the following exported functions to the deployment project:
   - `MPS_MVAR_demo_setup.m`
   - `mpsdemo_helper_split_scalar.m`
   - `mpsdemo_plot_mvar.m`
   - `mpsdemo_task_mvar.m`

   a. On the **Build** tab, in the **Exported Functions** area, click **Add files**.
   b. In the Add Files dialog box, browse to `$MPS_INSTALL\examples\montecarlo\`, select the exported functions listed above, and click **Open**.
4. Add the following helper files to the deployment project:
   - `mpsdemo_data_mvar.mat`

- `mpsdemo_help_getDefaults.m`
- `mpsdemo_setup_mvar.m`
- `mpsdemoconfig.m`
- `pTypeChecker.m`

a. On the **Build** tab, in the **Shared Resources and Helper Files** area, click **Add files/directories**.

b. In the Add Files dialog box, browse to *`$MPS_INSTALL`*`\examples\montecarlo` and select the helper files listed above and click **Open**.

5. In the Deployment Tool, click the Build icon (. When the build completes, you will find a file named `BondTools.ctf` in your project `distrib` 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 4-13 for more information.

2. Create a server using `mps-new`.

3. If you haven't 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 4-14 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 "Share the Deployable CTF Archive " on page 3-11 for complete details.

Step 4: Configure the Server for Concurrent Processing

By default, this Monte Carlo simulation runs four instances of `mpsdemo_task_mvar.m` at once. The default server configuration in `main_config` starts only one worker and one thread.

You now configure the server you have created to process this increased workload concurrently by increasing the number of configured workers and threads. You do this by modifying the server options `--num-workers` and `--num-threads` in the server configuration file, `main_config`.

1 Navigate to the folder containing the server instance you created in "Step 3: Share the Deployable CTF Archive on a Server" on page 5-52. 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 `--num-workers` and specify the value `4`. For example:

   ```
   --num-workers 4
   ```

4 Find the string `--num-threads` and specify the value `4`. For example:

   ```
   --num-threads 4
   ```

5 Save your changes to `main_config` and close the file.

6 Restart the server to retrieve the changes you made to `main_config` using `mps-restart`.

   From the server instance folder, issue the following command:

   ```
   mps-restart server_1
   ```

   where *`server_1`* is the server you created in "Step 3: Share the Deployable CTF Archive on a Server" on page 5-52. When you restart the server, the instance is stopped and started.

   For more information about `mps-restart` and related server commands, see the command reference pages in this documentation.

For more information about the server options `--num-workers` and `--num-threads`, see "How Does a Server Manage its Work?" on page 4-2

Step 5: Create the Java Client Code

Next, create a compatible client method interface to run the MATLAB code you have hosted on your 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 “MATLAB® Production Server™ Client Overview” on page 5-2, and “Java Client Coding Best Practices” on page 5-4.

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

Declare Java Method Signatures Compatible with MATLAB Functions You Deploy. In order to work with the MATLAB functions you defined in “Step 1: Write MATLAB Code” on page 5-49, declare the corresponding Java method signatures in the interface `MVARDemo`:

```
interface MVARDemo {

/** Loads simulation setup data into object array **/

    Object[] MPS_MVAR_demo_setup(int nArgOut) throws IOException,
                                                   MATLABException;

/** Subdivides simulation into discrete tasks **/

    Object[] mpsdemo_helper_split_scalar(int nArgOut, int intVal,
               int numTasks) throws IOException, MATLABException;

/** Generates plot on the server, sends it back to Java client
                                               as byte stream **/

    byte[] mpsdemo_plot_mvar(Object[] VaR, Object[] mVaR,
           double[] time, Object[] names) throws IOException,
                                               MATLABException;

/** Performs simulation to calculate mVaR of portfolio **/

    Object[] mpsdemo_task_mvar(int nArgOut, double splitTime,
           double[][] stock, double[] weights, double[] time,
            int numSims, double confLevel) throws IOException,
                                                    MATLABException;
}
```

Three of these signatures process multiple outputs; one does not. For a description of how each Java method signature maps to its MATLAB equivalent, see "Processing Outputs in the Monte Carlo Simulation" on page 5-26.

Instantiate MWClient, Create Proxy, and Specify Deployable CTF Archive. In the `main` method and `MPSTask` classes, you perform typical MATLAB Production Server client setup:

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

```
    public static void main(String[] args) {
          // Default settings
          MWClient client = null;
          MVARDemo mvarClient = null;
          try{
              client = new MWHttpClient();
....
```

2 Call `createProxy` on the new client instance, specifying the port number (`9910`) and CTF archive name (`mps_montecarlo_java`) the server is hosting in the `auto-deploy` folder:

```
...
mvarClient =
client.createProxy(new URL("http://user1.dhcp.mathworks.com:9910/BondTools"),
                                                            BondTools.class);
...
```

Cast MATLAB Arguments to Appropriate Java Types. In the `main` class, you cast arguments passed to an `Object` array called `output` from `MPS_MVAR_demo_setup` using compatible MATLAB and Java types.

For example, you cast a MATLAB `Double` (the default data type in MATLAB) to a Java `int`, because Java disallows float-point array indices:

```
int numTasks = ((Double) output[0]).intValue();
```

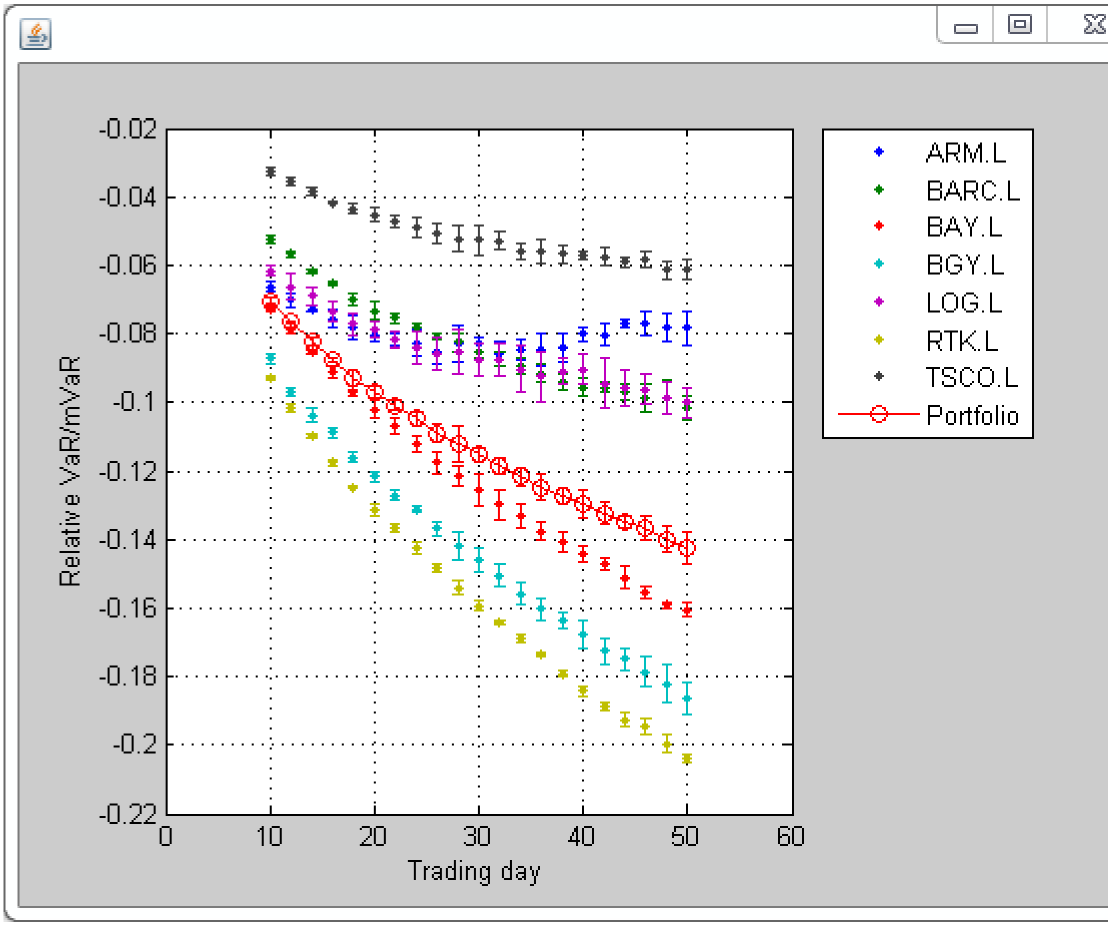
Step 6: Build the Client Code and Run the Example

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.

For more information, see "Java Client Prerequisites " on page 5-4.

When you run the simulation, if you use the default values provided with the example file, a trend graph is displayed as output:



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`.

The following table lists "Java Client Coding Best Practices" on page 5-4 and how the Java method signature conforms to these practices.

This Best Practice....	Is Illustrated By....
The method name exposed by the interface *must* match the name of the MATLAB function being deployed. The method and function must have the same number of inputs and outputs.	Both the MATLAB function signature and the Java method signature using the name `multipleOutputs`. Both signatures define two inputs and two outputs.
The method input and output types must be convertible to and from MATLAB.	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.

Note If you are passing an integer *as the first input argument* through a MATLAB function with multiple outputs defined, first wrap the integer in a `java.lang.Integer` object.

Processing Outputs in the Monte Carlo Simulation

In the “Monte Carlo Simulation for Java Client” on page 5-15 example, several types of method signatures are shown. The following sections compare how various types of outputs are handled with the Java client, in accordance with the guidelines described in “Code Multiple Outputs for Java Client” on page 5-24.

Multiple Inputs and Multiple Outputs. When comparing the following MATLAB and Java signatures, note the following:

- The Java signature accepts one more input parameter than the MATLAB function: `nArgOut`, specifying number of expected output arguments.
- The first input of the Java signature (an integer) which specifies the number of expected output arguments.

This Java signature....	Is the equivalent of this MATLAB signature....
`Object[] mpsdemo_helper_split_scalar(int nArgOut, int intVal, int numTasks) throws IOException, MATLABException;`	`function [integerPerTask, numTasks] = mpsdemo_helper_split_scalar(intVal, numTasks)`
`Object[] mpsdemo_task_mvar(int nArgOut, double splitTime, double[][] stock, double[] weights, double[] time, int numSims, double confLevel) throws IOException, MATLABException;`	`function [pVaR, mmVaR] = mpsdemo_task_mvar(numTimes, hVal, w, t, nSim, confLevel)`

Single Input and Multiple Outputs. When comparing the following MATLAB and Java signatures, note the following:

- The Java signature accepts a single integer input specifying number of expected output arguments (nArgOut).
- Java Object arrays represent cell arrays inside of MATLAB, as described in "Data Conversion with Java and MATLAB Types" on page 5-36 and the table "Conversion of MATLAB Types to Java Types " on page A-4.

This Java signature....	Is the equivalent of this MATLAB signature....
`Object[] MPS_MVAR_demo_setup(int nArgOut) throws IOException, MATLABException;`	`function [numTasks, numSims, numTimes, stock, names, weights, time, confLevel] = MPS_MVAR_demo_setup()`

Single Output Only. When comparing the following MATLAB and Java signatures, note the following:

- In contrast to the preceding signatures, this method call returns only *one* output of type byte array. Therefore, the signature does not need to specify the expected number of output arguments as the first input parameter.
- Java Object arrays represent cell arrays inside of MATLAB, as described in "Data Conversion with Java and MATLAB Types" on page 5-36 and the table "Conversion of MATLAB Types to Java Types " on page A-4.

This Java signature....	Is the equivalent of this MATLAB signature....
`byte[] mpsdemo_plot_mvar(Object[] VaR, Object[] mVaR, double[] time, Object[] names) throws IOException, MATLABException;`	`function image_data = mpsdemo_plot_mvar(VaR, mVaR, time, names)`

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`), the second input type is a `char` (`char2`), and the third input is a variable-length array that can contain zero, 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 above MATLAB function, is as follows:

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

In the `vararginout` method signature, equivalent Java types are specified for the two inputs (`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[]`.

The following table lists "Java Client Coding Best Practices" on page 5-4 and how the Java method signature conforms to these practices.

This Best Practice....	Is Illustrated By....
The method name exposed by the interface *must* match the name of the MATLAB function being deployed. The method and function must have the same number of inputs and outputs.	Both the MATLAB function signature and the Java method signature using the name `multipleOutputs`. Both signatures define two inputs and two outputs.
The method input and output types must be convertible to and from MATLAB.	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 (as shown in the Student class, above). 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 5-29 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.

If you compile `sortstudents` into a deployable CTF archive using the Deployment Tool (see “Create a Deployable CTF Archive from MATLAB Code” on page 3-6 for details) and make it available on the server at `http://localhost:9910/scoresorter` for access by the Java Client (see “Share the Deployable CTF Archive ” on page 3-11), you next define a Java class (`Student`) to represent the MATLAB structure, as follows:

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 (defined above).

Java Interface StudentSorter

```
interface StudentSorter {
    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 “Create a Java Application That Calls the Deployed Function” on page 2-10.

2. Create an unsorted student struct in Java that mimics the MATLAB struct in naming, number of inputs and outputs, and type validity in MATLAB. See “Java Client Coding Best Practices” on page 5-4 for more information.

3. Pass and display the unsorted student list using `System.out.println`.

4. Sort the student array and display it.

Java MPSClientExample 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;

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

public class MPSClientExample{

    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 ");
            System.out.println("the ascending order of scores :");
            for(Student s:sorted){
                System.out.println(s);
            }
        }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 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 the `Student` class is used as an output only, the data marshaling algorithm needs to create new instances of th class using the structure received from MATLAB.

This can be achieved using the `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

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

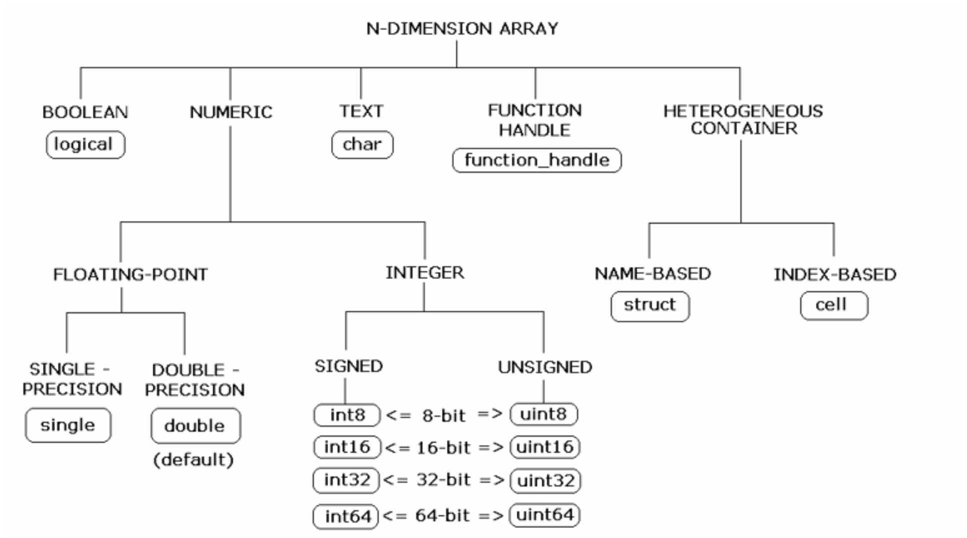
Working with MATLAB Data Types

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 5-37.

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 Classes

Detailed descriptions of expected conversion results for Java to MATLAB types and MATLAB types to Java, can be found in Appendix A, “Data Conversion Rules” .

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 by is performed automatically 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`

Truncating Dimensions in MATLAB and Java Data Conversion (Continued)

MATLAB Array Dimensions	Declared Output Java Type	Output Java Dimensions
`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 mechanisms 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

In this section...

.NET Client Coding Best Practices

When writing .NET interfaces for invoking MATLAB code, keep the following in mind:

- 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

Do the following 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 You should 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 "Create a Deployable CTF Archive from MATLAB Code" on page 3-6.

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

4 Your server should be running in order to run 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

Call the `close` method only when the `MWHttpClient` instance is no longer needed, as in the example interface above:

```
finally
{
   client.Close();
}
```

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 1:1 relationship at any point in time (for example, the server can not communicate with multiple clients simultaneously).

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

Using IDisposable to Free Client Instances. Call the `Dispose` method (an implementation of `IDisposable`) on unneeded client instances. Doing so ensures that all resources not already managed by the Garbage Collector (such as an instance of `MWClient`) are freed.

You call `Dispose` in either of two ways:

- **Call `Dispose` Directly** — Call the method directly on the object you want to free as follows, when the instance is no longer needed.:

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

  For an example of calling `Dispose` directly, see “Call Dispose to Free System Resources” on page 5-55 in the “Monte Carlo Simulation for .NET Client” on page 5-48 example.

- **Tie `IDispose` to the client instance with the `using` keyword** — When you instantiate `MWHttpClient`, you can attach `Dispose` for the lifetime of the client. If you do this, you don’t have to explicitly call `Dispose` on the instance—the garbage collector handles cleanup as it would any other managed resource.

  If you implement `IDisposable` in this manner, do so when you call `createProxy`, as in the following code snippet:

  ```
   using (MWClient client = new MWHttpClient(new TestConfigDispose()))
                {
                    DisposeTest proxy = client.CreateProxy(new
  Uri("http://localhost:" + server.Port));
  ```

In the MSDN documentation on `IDisposable.Dispose`, using a Finalizers is also mentioned as an alternative to the `using` keyword. This is not recommended for freeing instances of `MWClient`, and cannot be relied on to free resources in every case.

Note 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 "Create a .NET Application That Calls the Deployed Function" on page 2-14).

```
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 “Unsupported MATLAB Data Types for Client and Server Marshaling” on page 5-3

Where to Find the Ndoc

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, do the following 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, 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 (`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.

Monte Carlo Simulation for .NET Client

Monte Carlo simulations are useful for modeling systems whose inputs vary widely in range and scope. When working with such systems, it is often difficult to make accurate predictions due to the volatile nature of the data. In the financial industry, for example, the Monte Carlo method can be used to create realistic data sets used for trend analysis when working with fluctuating securities.

This example uses MATLAB Production Server to perform a Monte Carlo simulation on a number of stocks in a portfolio, yielding Value at Risk (*VaR*) and marginal value at risk (*mVaR*) at various confidence levels.

The values for *VaR* and *mVaR* are then plotted in a MATLAB figure.

This example shows how to create a Monte Carlo simulation that calculates *VaR* and *mVaR* concurrently using multiprocessing capabilities of the server and multithreading capabilities on the client.

Objectives

The Monte Carlo Simulation demonstrates the following:

- Invoking MATLAB functions concurrently with MATLAB Production Server
- Deploying MATLAB functions with multiple outputs
- Deploying a MATLAB function with graphical output

Where To Find the Example Code

All MATLAB and client code used to test and build client examples, can be found at `$MPS_INSTALL\examples`.

The example code listed in the documentation is not complete. Complete code is available in `$MPS_INSTALL\examples`, where `$MPS_INSTALL` is the location where you installed MATLAB Production Server.

Step 1: Write MATLAB Code

Before writing the client code, you write and deploy the Monte Carlo simulation code in MATLAB.

Modify the values in `mpsdemo_setup_mvar.m` as appropriate to the investment you want to model, or accept the default values in the program. Values include variables such as `numTimes`, `numSims`, and `time`. See the code comments for more information.

All files in this example can be found in `$MPS_INSTALL\examples\montecarlo\`.

This MATLAB code....	Provides this functionality....
`MPS_MVAR_demo_setup.m`	Prepares the Monte Carlo input data. Data is supplied in the associated MAT file (`mpsdemo_data_mvar.mat`).
`mpsdemoconfig.m`	Defines default configuration parameters used by the setup code (`MPS_MVAR_demo_setup.m`) such as number of tasks, number of simulations, and so on. One task is equivalent to one execution of `mpsdemo_task_mvar.m`.

This MATLAB code....	Provides this functionality....
`mpsdemo_setup_mvar.m`	Defines values related to the *mVar* calculations you want the Monte Carlo simulation to model (for example, number of simulations to perform, the number of times the simulations should be repeated, the stock price, proportionate weight of stock in portfolio and the confidence level at which we should calculate the value at risk). Calculates and returns level of difficulty used in the simulation.
`mpsdemo_helper_getDefaults.m`	Gets defaults from `mpsdemoconfig.m` to be used by `MPS_MVAR_demo_setup.m`.
`mpsdemo_helper_split_scalar.m`	Divides the simulation into concurrent tasks.
`mpsdemo_task_mvar.m`	Calculates *mVar* (marginal value at risk). One task is equivalent to one execution of `mpsdemo_task_mvar.m`. You set number of tasks by passing values to `mpsdemoconfig.m` (see following procedure).
`pTypeChecker.m`	Performs additional checks and calculations for `mpsdemo_task_mvar.m`.
`mpsdemo_plot_mvar.m`	Graphs the results of the Monte Carlo in a figure window.

Step 2: Create the Deployable CTF Archive that Runs the Simulation with the Deployment Tool

1 From MATLAB, start the Deployment Tool by entering `deploytool` using the MATLAB command prompt.

2 In the Deployment Project dialog box, create a project:

   a Enter `BondTools` in the **Name** field.

   b Select a location to store the project and enter it in the **Location** field.

   c In the **Type** drop-down, select **Generic CTF**.

   d Click **OK**.
   The Deployment Tool creates your Generic CTF target project.

3 Add the following exported functions to the deployment project:

   - `MPS_MVAR_demo_setup.m`
   - `mpsdemo_helper_split_scalar.m`
   - `mpsdemo_plot_mvar.m`
   - `mpsdemo_task_mvar.m`

   a On the **Build** tab, in the **Exported Functions** area, click **Add files**.

   b In the Add Files dialog box, browse to `*$MPS_INSTALL*\examples\montecarlo\`, select the exported functions listed above, and click **Open**.

4 Add the following helper files to the deployment project:

   - `mpsdemo_data_mvar.mat`
   - `mpsdemo_help_getDefaults.m`
   - `mpsdemo_setup_mvar.m`
   - `mpsdemoconfig.m`
   - `pTypeChecker.m`

   a On the **Build** tab, in the **Shared Resources and Helper Files** area, click **Add files/directories**.

   b In the Add Files dialog box, browse to `*$MPS_INSTALL*\examples\montecarlo` and select the helper files listed above and click **Open**.

5 In the Deployment Tool, click the Build icon (. When the build completes, you will find a file named `BondTools.ctf` in your project `distrib` 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 4-13 for more information.

2. Create a server using `mps-new`.

3. If you haven't 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 4-14 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 "Share the Deployable CTF Archive " on page 3-11 for complete details.

Step 4: Configure the Server for Concurrent Processing

By default, this Monte Carlo simulation runs four instances of `mpsdemo_task_mvar.m` at once. The default server configuration in `main_config` starts only one worker and one thread.

You now configure the server you have created to process this increased workload concurrently by increasing the number of configured workers and threads. You do this by modifying the server options `--num-workers` and `--num-threads` in the server configuration file, `main_config`.

1. Navigate to the folder containing the server instance you created in "Step 3: Share the Deployable CTF Archive on a Server" on page 5-52. 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 `--num-workers` and specify the value `4`. For example:

   ```
   --num-workers 4
   ```

4. Find the string `--num-threads` and specify the value `4`. For example:

```
--num-threads 4
```

5 Save your changes to `main_config` and close the file.

6 Restart the server to retrieve the changes you made to `main_config` using `mps-restart`.

   From the server instance folder, issue the following command:

   ```
   mps-restart server_1
   ```

   where *`server_1`* is the server you created in "Step 3: Share the Deployable CTF Archive on a Server" on page 5-52. When you restart the server, the instance is stopped and started.

   For more information about `mps-restart` and related server commands, see the command reference pages in this documentation.

For more information about the server options `--num-workers` and `--num-threads`, see "How Does a Server Manage its Work?" on page 4-2

Step 5: Create the C# Client Code

Next, create a compatible client method interface to run the MATLAB code you have hosted on your server.

When developing your C# code, note the following essential tasks, described in the sections that follow. For more information about clients coding basics and best practices, see "MATLAB® Production Server™ Client Overview" on page 5-2, and ".NET Client Coding Best Practices" on page 5-43.

This documentation references specific portions of the client code. You can find the complete C# client code in *`$MPS_INSTALL`*`\examples\montecarlo`.

Declare C# Method Signatures Compatible with MATLAB Functions You Deploy. In order to work with the MATLAB functions you defined in "Step 1: Write MATLAB Code" on page 5-49, declare the corresponding C# method signatures in the interface `MVaRDemo` in `marginalvalueatrisk.cs`:

```
public interface MVaRDemo
    {
```

```
/** Loads simulation data - computes the difficulty and
                          returns it as part of setup **/

 void MPS_MVAR_demo_setup(out double numTasks,
      out double numSims, out double numTimes,
      out double[,] stock, out object[] names,
      out double[] weights, out double[] time,
      out double confLevel);

/** Subdivides simulation into discrete tasks **/

  void mpsdemo_helper_split_scalar(out double[] splitTimes,
      out int numTasksOut, int intVal, int numTasks);

/** Generates plot on the server, sends it back to
                               .NET client as sbyte stream **/

 sbyte[] mpsdemo_plot_mvar(object[] VaR, object[] mVaR,
                         double[] time, object[] names);

/** Performs simulation to calculate mVaR of portfolio **/

 void mpsdemo_task_mvar(out object[] VarVals, out object[] mVarVals,
      double splitTime, double[,] stock, double[] weights,
      double[] time, int numSims, double confLevel);    }
```

Three of these signatures process multiple outputs; one does not. For a description of how each C# method signature maps to its MATLAB equivalent, see “Processing Multiple Outputs in the Monte Carlo Simulation” on page 5-59.

Instantiate MWClient, Create Proxy, and Specify Deployable CTF Archive. In the `main` method and `MPSTask` classes, you perform typical MATLAB Production Server client setup:

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

```
....
MWClient client = null;
            MVaRDemo mvarClient = null;
            try
            {
                client = new MWHttpClient();
....
```

2 Call `createProxy` on the new client instance, specifying the port number (`9910`) and CTF archive name (`BondTools`) the server is hosting in the `auto-deploy` folder.

   The .NET code references the server and port through a constructor named `SERVER`:

```
private const string SERVER = @"http://user-01.dhcp.mathworks.com:9930/BondTools";

try
            {
                client = new MWHttpClient();
                mvarClient = client.CreateProxy(new Uri(SERVER));...
```

Call Dispose to Free System Resources. The `Dispose` method is called in both `CallMPS` and `Main`, to ensure system resources are freed.

For more about calling `Dispose`, see "Using IDisposable to Free Client Instances" on page 5-45.

```
finally
            {
                client.Dispose();
            }
```

Step 6: Build the Client Code and Run the Example

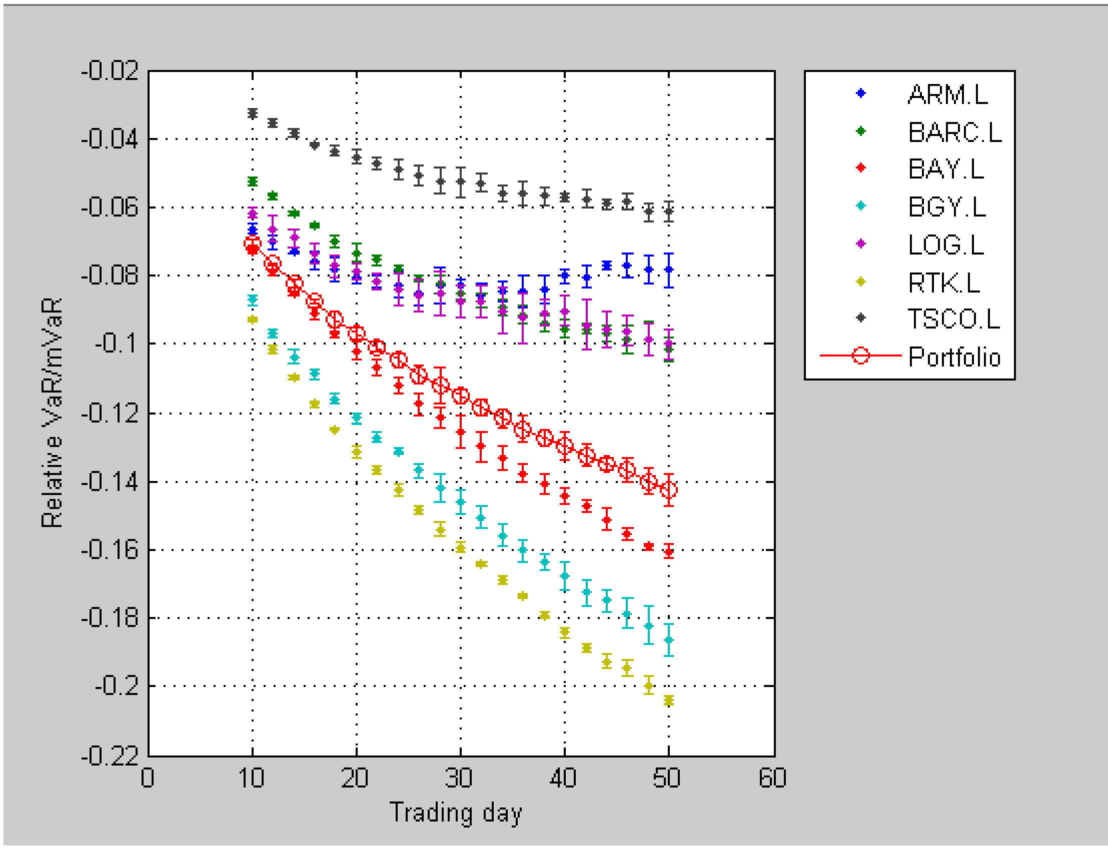
Before you attempt to build and run your client code, ensure that you have done the following:

- Added `Mathworks.MATLAB.ProductionServer.Client.dll` (*`$MPS_INSTALL`*`\client\net`) to your Microsoft Visual Studio project references.
- 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.

For more information, see ".NET Client Prerequisites" on page 5-43.

When you run the simulation, if you use the default values provided with the example file, a trend graph is displayed as output:



Code Multiple Outputs for C# .NET Client

MATLAB allows users to write functions with multiple outputs. In order to work with 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 calculation.

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

To deploy this function with MATLAB Production Server, you need to write a corresponding method interface in C#, using the `out` keyword. The `out` keyword causes arguments to be passed by reference. When using `out`, both the interface method definition and the calling method must 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 snippets listed here, multiple outputs are listed (with matching specified data types) in the same order as 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");

    output1 = mpsexample.multipleoutputs;
       Console.ReadLine();
}
```

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`.

The following table lists ".NET Client Coding Best Practices" on page 5-43 and how the C# interface method signature conforms to these practices.

This Best Practice....	Is Illustrated By....
The method name exposed by the interface *must* match the name of the MATLAB function being deployed. The method and function must have the same number of inputs and outputs.	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.
The method input and output types must be convertible to and from MATLAB.	MATLAB .NET 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.

Processing Multiple Outputs in the Monte Carlo Simulation

In the "Monte Carlo Simulation for .NET Client" on page 5-48 example, several types of method signatures are shown. The following sections compare how various types of outputs are handled with the .NET client, in C#, in accordance with the guidelines described in "Code Multiple Outputs for C# .NET Client" on page 5-57.

Multiple and SIngle Inputs with Multiple Outputs. When comparing the following MATLAB and C# signatures, note the following:

- The C# signature explicitly specifies the `out` keyword to designate an output.
- Multiple outputs listed in the C# signature match the outputs in the MATLAB signature exactly. For example, in the C# signature `mpsdemo_helper_split_scalar`, `splitTimes` corresponds to the MATLAB argument `integerPerTask`. `numTasksOut` corresponds to `numTasks`. Note that these corresponding outputs are listed in the same order in both signatures. Notice the data types in the C# signature match exactly to the MATLAB data types and are explicitly specified in the C# code.

This C# signature....	Is the equivalent of this MATLAB signature....
`void mpsdemo_helper_split_scalar(out double[] splitTimes, out int numTasksOut, int intVal, int numTasks);`	`function [integerPerTask, numTasks] = mpsdemo_helper_split_scalar(intVal, numTasks)`
`void mpsdemo_task_mvar(out object[] VarVals, out object[] mVarVals, double splitTime, double[,] stock, double[] weights, double[] time, int numSims, double confLevel);`	`function [pVaR, mmVaR] = mpsdemo_task_mvar(numTimes, hVal, w, t, nSim, confLevel)`
`void MPS_MVAR_demo_setup(out double numTasks, out double numSims, out double numTimes, out double[,] stock, out object[] names, out double[] weights, out double[] time, out double confLevel);`	`function [numTasks, numSims, numTimes, stock, names, weights, time, confLevel] = MPS_MVAR_demo_setup()`

Single Output Only. When comparing the following MATLAB and C# signatures, note the following:

- In contrast to the preceding signatures, this method call returns only *one* output of type `sbyte` array. Therefore, the signature does not need to specify the `out` keyword to denote multiple outputs.
- Object arrays represent cell arrays inside of MATLAB, as described in "Data Conversion with C# and MATLAB Types" on page 5-74 and the table "Conversion Between MATLAB Types and C# Types" on page A-6.

This C# signature....	Is the equivalent of this MATLAB signature....
`sbyte[] mpsdemo_plot_mvar(object[] VaR, object[] mVaR, double[] time, object[] names);`	`function image_data = mpsdemo_plot_mvar(VaR, mVaR, time, names)`

Code Variable-Length Inputs and Outputs for .NET Client

MATLAB Production Server .NET client supports MATLAB's ability to work with variable-length inputs. See the *MATLAB Function Reference* for complete information on `varargin` and `varargout`.

Using varargin with .NET Client

MATLAB variable input arguments (`varargin`) are passed using the `params` keyword.

For example, consider the MATLAB function `varargintest`, which takes a variable-length input (`varargin`) — containing strings and integers — and returns all 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 `VararginTest`, 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();
}
```

The following table lists ".NET Client Coding Best Practices" on page 5-43 and how the C# interface method signature conforms to these practices.

This Best Practice....	Is Illustrated By....
The method name exposed by the interface *must* match the name of the MATLAB function being deployed. The method and function must have the same number of inputs and outputs.	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
The method input and output types must be convertible to and from MATLAB.	MATLAB .NET interface supports direct conversion between MATLAB cell arrays and C# object arrays. See "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 dimensions.

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 5-57), 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 5-61, 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");
}
```

The following table lists ".NET Client Coding Best Practices" on page 5-43 and how the C# interface method signature conforms to these practices.

This Best Practice....	Is Illustrated By....
The method name exposed by the interface *must* match the name of the MATLAB function being deployed. The method and function must have the same number of inputs and outputs.	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,
The method input and output types must be convertible to and from MATLAB.	MATLAB .NET interface supports direct conversion of MATLAB strings and integers. See "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

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`.

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`.

  .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;
            }
        }
    }
```

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 ".NET Type Conversion Requirements To and From a MATLAB Structure" on page 5-70 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.

C# Interface StudentSorter

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

C# sortstudents Method

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

    Student s1 = new Student();
    s1.name = "Tony Miller";
    s1.score = 91;
    s1.grade = "A-";

    Student s2 = new Student();
    s2.name = "Ed Plum";
    s2.score = 83;
    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 = mpsexample.sortstudents(unsorted);

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

    Console.ReadLine();
}
```

.NET Type Conversion Requirements To and From a MATLAB Structure.

Input To and Output From MATLAB Structures Using Fields and Properties

Before you can successfully marshal .NET types to and from MATLAB structures, as inputs and outputs, ensure your .NET struct or class meets the following requirements.

Output from MATLAB Structures Using a Constructor

Using Attributes

In addition to using the techniques described in "Using .NET Structs and Classes" on page 5-66, 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 5-72
- "Interface Attributes" on page 5-72

- "Fields and Property Attributes" on page 5-72

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

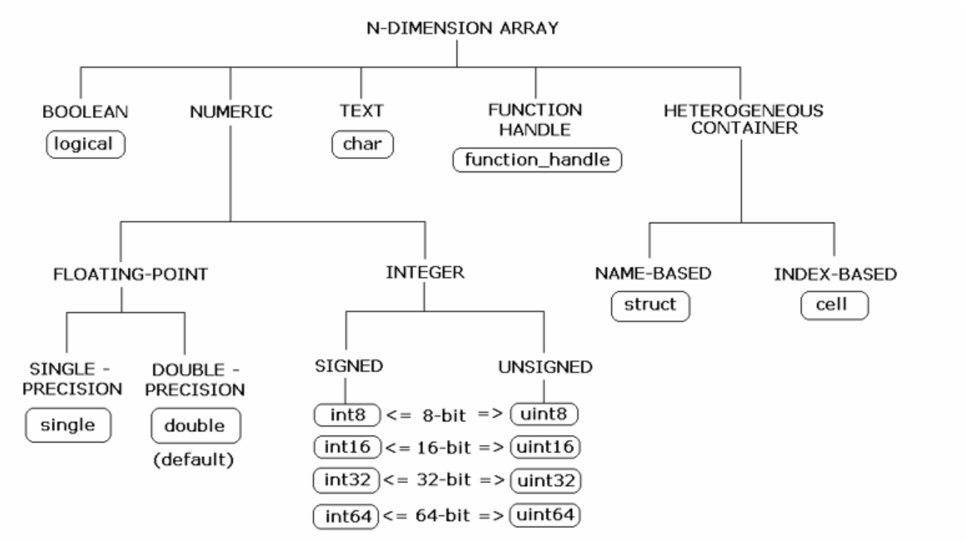
When the .NET client invokes a MATLAB function through a request and receives a result in the response, data conversion takes place between MATLAB types and C# types.

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 5-75.

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".

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();
```

6

Commands — Alphabetical List

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

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

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 4-9

Concepts

- “License Management for MATLAB® Production Server™” on page 4-8

mps-new

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 "Server Instance" on page 6-4.

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

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

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-start` | `mps-status`

Related Examples

- "Create a Server" on page 4-11

Concepts

- "Server Creation" on page 4-10
- "Server Overview" on page 4-2

mps-restart

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`

mps-start

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 4-19

Concepts

- "Server Startup" on page 4-18
- "Server Overview" on page 4-2

mps-status

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 4-19

Concepts

- “Server Startup” on page 4-18
- “Server Overview” on page 4-2

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-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