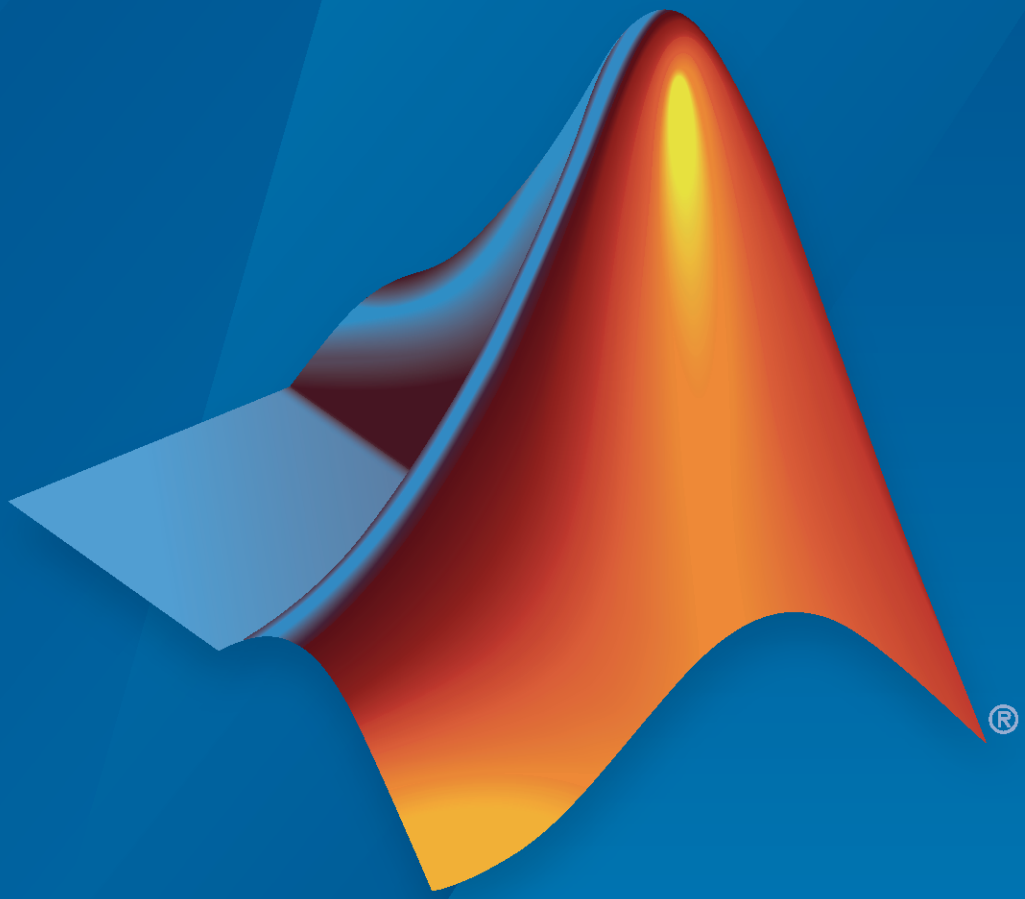
MATLAB® Production Server™

.NET Programming Guide



MATLAB®

R2020a

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

MATLAB® Production Server™ .NET Programming Guide

Revision History

March 2014	Online only	New for Version 1.2 (Release R2014a)
October 2014	Online only	Revised for Version 2.0 (Release R2014b)
March 2015	Online only	Revised for Version 2.1 (Release R2015a)
September 2015	Online only	Revised for Version 2.2 (Release R2015b)
March 2016	Online only	Revised for Version 2.3 (Release 2016a)
September 2016	Online only	Revised for Version 2.4 (Release 2016b)
March 2017	Online only	Revised for Version 3.0 (Release 2017a)
September 2017	Online only	Revised for Version 3.0.1 (Release R2017b)
March 2018	Online only	Revised for Version 3.1 (Release R2018a)
September 2018	Online only	Revised for Version 4.0 (Release R2018b)
March 2019	Online only	Revised for Version 4.1 (Release R2019a)
September 2019	Online only	Revised for Version 4.2 (Release R2019b)
March 2020	Online only	Revised for Version 4.3 (Release R2020a)

Contents

1 Client Programming

2 .NET Client Programming

Client Programming

MATLAB Production Server Examples

Additional Client examples for MATLAB Production Server are available in the `client` folder of your MATLAB Production Server.

Create a .NET MATLAB Production Server Client

To create a MATLAB Production Server client:

1. Obtain the client run-time files installed in *$MPS_INSTALL*/`client`.
2. In consultation with the MATLAB programmer, agree on the MATLAB function signatures that comprise the services in the application.
3. Configure your system with the appropriate software for working with .NET.

   See “Prepare Your Microsoft Visual Studio Environment” on page 2-4.
4. Based on your requirements, decide if the client uses a static proxy or a dynamic proxy.

   - A static proxy uses an object implementing an interface that mirrors the deployed MATLAB functions. You provide the interface for the static proxy.

     See “Static Proxy Interface Guidelines” on page 2-2.
   - A dynamic proxy creates server requests based on the MATLAB function name provided to the `invoke()` method. You provide the function name, the number of output arguments, and all of the input arguments required to evaluate the functions.

     See “Invoke MATLAB Functions Dynamically” on page 2-7.
5. Write a the .NET code to instantiate a proxy to a MATLAB Production Server instance and call the MATLAB functions.

   a. Create a dynamic proxy for communicating with the service hosted by MATLAB Production Server software.
   b. Declare and throw exceptions as required.
   c. Free system resources using the `close` method of `MWClient`, after making needed calls to your application.

Create a C# Client

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

In your C# code, you must:

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

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

Create a Microsoft Visual Studio Project

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

Create a Reference to the Client Run-Time Library

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

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

Design the .NET Interface in C#

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

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

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

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

might look like this:

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

Note the following:

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

Write, Build, and Run the .NET Application

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

1. Open the Microsoft Visual Studio project, `MagicSquare`, that you created earlier.
2. In `Program.cs` tab, paste in the code below.

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

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

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

namespace Magic
{
     public class MagicClass
     {

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

    public static void Main(string[] args)
     {
        MWClient client = new MWHttpClient();
      try
      {
          Magic me = client.CreateProxy<Magic>
                    (new Uri("http://localhost:9910/mymagic_deployed"));
            double[,] result1 = me.mymagic(4);
            print(result1);
      }
      catch (MATLABException ex)
      {
        Console.WriteLine("{0} MATLAB exception caught.", ex);
        Console.WriteLine(ex.StackTrace);
      }
```

```
        catch (WebException ex)
        {
          Console.WriteLine("{0} Web exception caught.", ex);
          Console.WriteLine(ex.StackTrace);
        }
        finally
        {
          client.Dispose();
        }
        Console.ReadLine();
    }

    public static void print(double[,] x)
    {
       int rank = x.Rank;
       int [] dims = new int[rank];

       for (int i = 0; i < rank; i++)
       {
          dims[i] = x.GetLength(i);
       }

       for (int j = 0; j < dims[0]; j++)
       {
          for (int k = 0; k < dims[1]; k++)
          {
             Console.Write(x[j,k]);
             if (k < (dims[1] - 1))
             {
                Console.Write(",");
             }
          }
          Console.WriteLine();
       }
     }
    }
}
```

3 Build the application. Click **Build > Build Solution**.

4 Run the application. Click **Debug > Start Without Debugging**. The program returns the following console output:

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

Unsupported MATLAB Data Types for Client and Server Marshaling

Supported Data Types

MATLAB Production Server supports marshaling of the following MATLAB data types between client applications and server instances.

- Numeric types – Integers and floating-point numbers
- Character arrays
- Structures
- Cell arrays
- Logical

Unsupported Data Types

Following are just a few examples MATLAB data types that MATLAB Production Server does not support for marshaling between the server and the client.

- MATLAB function handles
- Sparse matrices
- Tables
- Timetables
- Complex numbers

See Also

More About

- “JSON Representation of MATLAB Data Types”

.NET Client Programming

.NET Client Coding Best Practices

Static Proxy Interface Guidelines

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

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

.NET Client Prerequisites

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

1. Install Microsoft Visual Studio. See https://www.mathworks.com/support/compilers/current_release/ for an up-to-date listing of supported software, including IDEs and Microsoft .NET Frameworks.
2. Verify that your application is deployed to a running server instance.

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

The connections between a .NET client and the servers with which it interacts are managed by one or more instances of `MWHttpClient`. You can use a single instance to communicate with more than one server or you can create multiple instances to manage multiple servers. Proxy objects, created using an instance of `MWHttpClient`, communicate with the server until the `Dispose` method of that instance is invoked. Therefore, it is important to call the `Dispose` method only when the `MWHttpClient` instance is no longer needed, to reclaim system resources.

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

You call `Dispose` in either of two ways:

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

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

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

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

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

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

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 1-7

Where to Find the API Documentation

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

Prepare Your Microsoft Visual Studio Environment

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

Create a Microsoft Visual Studio Project

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

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, `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.

Configure the Client-Server Connection

In this section...

You configure the client-server connection using an object that implements the `MWHttpClientConfig` interface. This interface defines these properties:

- `TimeoutMilliSeconds` determines the amount of time, in milliseconds, the client waits for a response before timing out
- `ResponseSizeLimit` determines the maximum size, in bytes, of the response a client accepts.

The API provides a default implementation, `MWHttpClientDefaultConfig`, that is automatically used when an HTTP client is instantiated. To modify the configuration, extend `MWHttpClientDefaultConfig` and pass it to the HTTP client constructor.

Create a Connection with the Default Configuration

When you create a client connection using the default constructor, `MWHttpClient()`, an instance of `MWHttpClientDefaultConfig` is automatically used to configure the client-server connection. The default configuration sets these connection properties:

- `TimeOutMs = 120000`
- `ResponseSizeLimit = 64*1024*1024` (64 MB)

Create a Connection with a Custom Configuration

To change one or more connection properties:

1. Implement a custom connection configuration by extending the `MWHttpClientDefaultConfig` interface.
2. Create the client connection using one of the constructors that accepts a configuration object.
   - `MWHttpClient(MWHttpClientConfig config)`
   - `MWHttpClient(MWHttpClientConfig config, MWSSLConfig securityConfig)`

This code sample creates a client connection with a timeout value of 1000 ms:

```
class MyClientConfig : MWHttpClientDefaultConfig
{
  public override int TimeoutMilliSeconds
  {
    get { return 1000; }
  }
}
...
MWClient client = new MWHttpClient(new MyClientConfig());
...
```

Implementing a Custom Connection Configuration

To implement a custom connection configuration extend the `MWHttpClientDefaultConfig` interface. The `MWHttpClientDefaultConfig` interface has one getter method for each configuration property.

To specify that a client times out after 1 s and can only accept 4 MB responses:

```
class MyClientConfig : MWHttpClientDefaultConfig
{
  public override int TimeoutMilliSeconds
  {
    get { return 60000; }
  }
  public override int ResponseSizeLimit
  {
    get { return 4*1024*1024; }
  }
}
```

Invoke MATLAB Functions Dynamically

In this section...

To dynamically invoke MATLAB functions, specify the function name as one of the parameters to the method invoking the request. You do not need to create a compiled interface that models the contents of a deployable archive, nor do you have to change your client application if there are changes to functions in the deployable archive.

To dynamically invoke a MATLAB function:

1. Instantiate an `MWClient` instance.
2. Create a reflection-based proxy object using one of the `CreateComponentProxy()` methods of the client connection.
3. Invoke the function, or functions, using one of the `Invoke()` methods of the reflection-based proxy.

Create a Reflection-Based Proxy

A reflection-based proxy implements the `MWInvokable` interface and provides methods that allow you directly invoke any MATLAB function deployed as part of a deployable archive. As with the interface-based proxy, a reflection-based proxy is created from an `MWClient` object. The `MWClient` interface has two methods for creating a reflection-based proxy:

- `MWInvokable CreateComponentProxy(URL archiveURL)` creates a proxy that uses standard MATLAB data types.
- `MWInvokable CreateComponentProxy(URL archiveURL, MWMarshallingRules marshallingRules)` creates a proxy that uses structures.

To create a reflection-based proxy for invoking functions in the `myMagic` archive, hosted on your local computer:

```
MWClient myClient = new MWHttpClient();

Uri archiveURL = new Uri("http://localhost:9910/myMagic");
MWInvokable myProxy =  myClient.CreateComponentProxy(archiveURL);
```

Invoke a MATLAB Function with a Dynamic Proxy

A dynamic proxy has three methods for invoking functions on a server:

- `Object[] Invoke(string functionName, IList<Type> targetTypes, params Object[] inputs)` invokes a function that returns multiple values.
- `T Invoke<T>(string functionName, params Object[] inputs)` invokes a functions that returns a single value.
- `void Invoke(string functionName, params Object[] inputs)` invokes a function that returns no values.

All of the methods map to the MATLAB function as follows:

- First argument is the function name
- Last arguments are the function inputs

Return Multiple Outputs

The MATLAB function `myLimits` returns two values.

```
function [myMin,myMax] = myLimits(myRange)
 myMin = min(myRange);
 myMax = max(myRange);
end
```

To invoke `myLimits` from a .NET client, use the `Invoke()` method that takes a list of target types:

```
double[] myRange = new double[]{2,5,7,100,0.5};
IList<Type> targetTypes =
    new List<Type> { typeof(double), typeof(double) };
Object[] myLimits = myProxy.Invoke("myLimits", targetTypes, myRange);
double myMin = myLimits[0];
double myMax = myLimits[1];
Console.WriteLine("min: {0:f} max: {1:f}", myMin, myMax);
```

This form of `Invoke()` always returns `Object[]`. The contents of the returned array are typed based on the contents of `targetType`.

Return a Single Output

The MATLAB function `addmatrix` returns a single value.

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

To invoke `addmatrix` from a .NET client, use the `Invoke()` method that does not take the number of return arguments:

```
double[,] a1 = new double[,] {{1,2,3},{3,2,1}};
double[,] a2 = new double[,] {{4,5,6},{6,5,4}};
Object[] inputs = new Object[2];
inputs[0] = a1;
inputs[1] = a2;
double[,] result = myProxy.Invoke<double[,]>("addmatrix", inputs);

// display the result
```

Return No Outputs

The MATLAB function `foo` does not return a value.

```
function foo(a1)
min(a1);
```

To invoke `foo` from a .NET client, use the `Invoke()` method that returns void:

```
double[,] a = new double [,] {{1,2,3},{3,2,1}};
myProxy.Invoke("foo", a);
```

Create Custom Marshaling Rules

You need to provide custom marshaling rules to the reflection-based proxy if:

- any MATLAB function in a deployable archive uses structures
- any MATLAB in a deployable archive requires a custom setting to the default marshaling rules.

  There are default rules marshaling `NaN`, `DateTime`, .NET null, 1xN vectors, and Nx1 vectors.

To provide marshaling rules to the proxy:

1. Implement a new set of marshaling rules by extending `MWDefaultMarshalingRules` to override the defaults.
2. Create the proxy using `CreateComponentProxy(URL archiveURL, MWMarshallingRules marshalingRules)`.

The deployable archive `studentCheck` includes functions that use a MATLAB structure of the form

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

Client code represents the MATLAB structure with a class named `Student`. To create a marshaling rule for dynamically invoking the functions in `studentChecker`, create a class named `studentMarshaller`.

```
class studentMarshaler : MWDefaultMarshalingRules
{
  public override IList<Type> StructTypes()
  {
    get { return new List<Type> { typeof(Student) }; }
  }
}
```

Create the dynamic proxy for `studentCheck` by passing `studentMarshaller` to `createComponentProxy()`.

```
URL archiveURL = new URL("http://localhost:9910/studentCheck");
myProxy =  myClient.CreateComponentProxy(archiveURL,
                                         new StudentMarshaler());
```

For more information about using MATLAB structures, see “Marshal MATLAB Structures (structs) in C#” on page 2-20.

For more information about the other data marshaling rules, see “Data Conversion with C# and MATLAB Types” on page 2-28.

Access Secure Programs Using HTTPS

In this section...

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

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

Configure the Client Environment for SSL

At a minimum the client requires the server's root CA (Certificate Authority) in one of the application's certificate stores.

To connect to a server that requires client-side authentication, the client needs a signed certificate in one of the application's certificate stores.

To manage the client’s certificates, use `makecert`.

Establish a Secure Proxy Connection

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

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

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

Establish a Secure Connection Using Client Authentication

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

1. Provide an implementation of the `MWSSLConfig` interface that returns a valid client certificate collection.
2. Use the `MWHttpClient` constructor that takes an instance of your `MWSSLConfig` implementation to create the connection to the server instance.
3. Create the proxy using the program's `https://` URL.

Implement the MWSSLConfig Interface

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

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

Establish the Secure Connection

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

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

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

Implement Advanced Authentication Features

The .NET `ServicePointManager.ServerCertificateValidationCallback` property allows you add extra layers of security to:

- Disable SSL protocols to protect against the POODLE exploit.

  ```
  System.Net.ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls | SecurityProtocolT
  MWClient client = new MWHttpClient();
  ```

- Perform alternate hostname verification to authenticate servers when the URL hostname does not match the certificate's hostname
- Ensure that the client shares data only with specific servers

The `ServerCertificateValidationCallback` property is a delegate that processes the certificates during the SSL handshake. By default, no delegate is implemented, so no custom processing is performed. You can provide an implementation to perform any custom authorization required.

See Also

External Websites

- MSDN documentation
- .Net `ServicePointManager` documentation

Bond Pricing Tool for .NET Client

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

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

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

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

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

Objectives

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

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

Step 1: Write MATLAB Code

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

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

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

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

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

end
```

Step 2: Create a Deployable Archive with the Production Server Compiler App

To create the deployable archive for this example:

1. From MATLAB, select the Production Server Compiler app.
2. In the **Application Type** list, select **Deployable Archive**.
3. In the **Exported Functions** field, add `pricecalc.m`.

`pricecalc.m` is located in *`MPS_INSTALL`*`\client\dotnet\examples\BondPricingTool\MATLAB`.

4. Under **Application Information**, change `pricecalc` to `BondTools`.
5. Click **Package**.

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

Step 3: Share the Deployable Archive on a Server

1. Download the MATLAB Runtime, if needed, at https://www.mathworks.com/products/compiler/mcr. See "Download and Install the MATLAB Runtime" for more information.
2. Create a server using `mps-new`. See "Create a Server" for more information.
3. If you have not already done so, specify the location of the MATLAB Runtime to the server by editing the server configuration file, `main_config` and specifying a path for `--mcr-root`. See "Edit the Configuration File" 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 Archive" for complete details.

Step 4: Create the C# Client Code

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

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

Declare C# Method Signatures Compatible with MATLAB Functions You Deploy

To use the MATLAB functions you defined in "Step 1: Write MATLAB Code" on page 2-12, declare the corresponding C# method signature in the interface `BondTools.cs`:

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

This interface creates an array of primitive `double` types, corresponding to the MATLAB primitive types (`Double`, in MATLAB, unless explicitly declared) in `pricecalc.m`. A one to one mapping exists between the input arguments in both the MATLAB function and the C# interface The interface specifies compatible type `double`. This compliance between the MATLAB and C# signatures demonstrates the guidelines listed in ".NET Client Coding Best Practices" on page 2-2.

Instantiate MWClient, Create Proxy, and Specify Deployable Archive

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

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

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

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

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

Use Dispose() Consistently to Free System Resources

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

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

For example, note that in `ServerBondToolsFactory.cs`, you dispose of the `MWHttpClient` instance you created in “Instantiate MWClient, Create Proxy, and Specify Deployable Archive” on page 2-13 when it is no longer needed.

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

`Dispose()` is an implementation of `IDisposable`. For more information about using `Dispose()` to free resources, see “Use Dispose() Consistently to Free System Resources” on page 2-14.

Step 5: Build the Client Code and Run the Example

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

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

Code Multiple Outputs for C# .NET Client

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

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

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

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

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

MATLAB Function multipleoutputs

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

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

C# Interface MultipleOutputsExample

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

C# Method TryMultipleOutputs

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

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

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

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

Note the following coding best practices illustrated by this example:

- Both the MATLAB function signature and the C# interface method signature use the name `multipleOutputs`. Both MATLAB and C# code are processing three inputs and three outputs.
- MATLAB .NET interface supports direct conversion from C# `double` array to MATLAB `double` array and from C# `string` to MATLAB char array. For more information, see “Data Conversion with C# and MATLAB Types” on page 2-28 and “Conversion Between MATLAB Types and C# Types” on page A-2.

Code Variable-Length Inputs and Outputs for .NET Client

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

Using varargin with .NET Client

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

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

Example 2.1. 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`.

Example 2.2. 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.

Example 2.3. C# Method TryVarargin

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

Note the following coding best practices illustrated by this example:

- Both the MATLAB function signature and the C# interface method signature use the name `varargintest`. Both MATLAB and C# code are processing two variable-length inputs, string and integer.

- MATLAB .NET interface supports direct conversion between MATLAB cell arrays and C# object arrays. See “Data Conversion with C# and MATLAB Types” on page 2-28 and “Conversion Between MATLAB Types and C# Types” on page A-2 for more information.

Using varargout with .NET Client

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

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

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

Example 2.4. 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 2-15), a `double` input (`in1`) and a `char` 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 2-17, you use the C# `params` keyword to pass the variable-length input.

Example 2.5. 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`).

Example 2.6. C# Method TryVarargout

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

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

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

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

Note the following coding best practices illustrated by this example:

- Both the MATLAB function signature and the C# interface method signature use the name `varargouttest`. Both MATLAB and C# code are processing a variable-length input, a variable-length output, and two multiple non-variable-length outputs.
- MATLAB .NET interface supports direct conversion between MATLAB cell arrays and C# object arrays. See “Data Conversion with C# and MATLAB Types” on page 2-28 and “Conversion Between MATLAB Types and C# Types” on page A-2 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 `struct`s and classes.

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

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

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

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

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

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

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

Using .NET Structs and Classes

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

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

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

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

.NET Struct Student

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

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

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

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

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

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

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

C# Class SimpleStruct

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

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

     public string Field1
     {
         get
         {
             return f1;
         }
     }
```

```
    public double Field2
    {
        get
        {
            return f2;
        }
    }
}
```

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

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

Your .NET structs and classes must adhere to specific requirements, based on both the level of scoping (fields and properties as opposed to constructor, for example) and whether you are marshaling .NET types to or from a MATLAB structure. See “Using .NET Structs and Classes” on page 2-20 for details.

MATLAB Function sortstudents

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

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

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

C# Interface StudentSorter

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

C# Class ClientExample

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

namespace MPS
```

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

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

            try
            {
                client = new MWHttpClient();
                StudentSorter mpsexample =
                    client.CreateProxy(new Uri("http://test-machine:9910/scoresorter"));

                Student s1 = new Student();
                s1.name = "Tony Miller";
                s1.score = 90;
                s1.grade = "A";

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

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

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

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

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

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

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

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

            }
            catch (MATLABException ex)
            {
```

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

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

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

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

Using Attributes

In addition to using the techniques described in “Using .NET Structs and Classes” on page 2-20, 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 2-25
- “Interface Attributes” on page 2-25
- “Fields and Property Attributes” on page 2-26

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

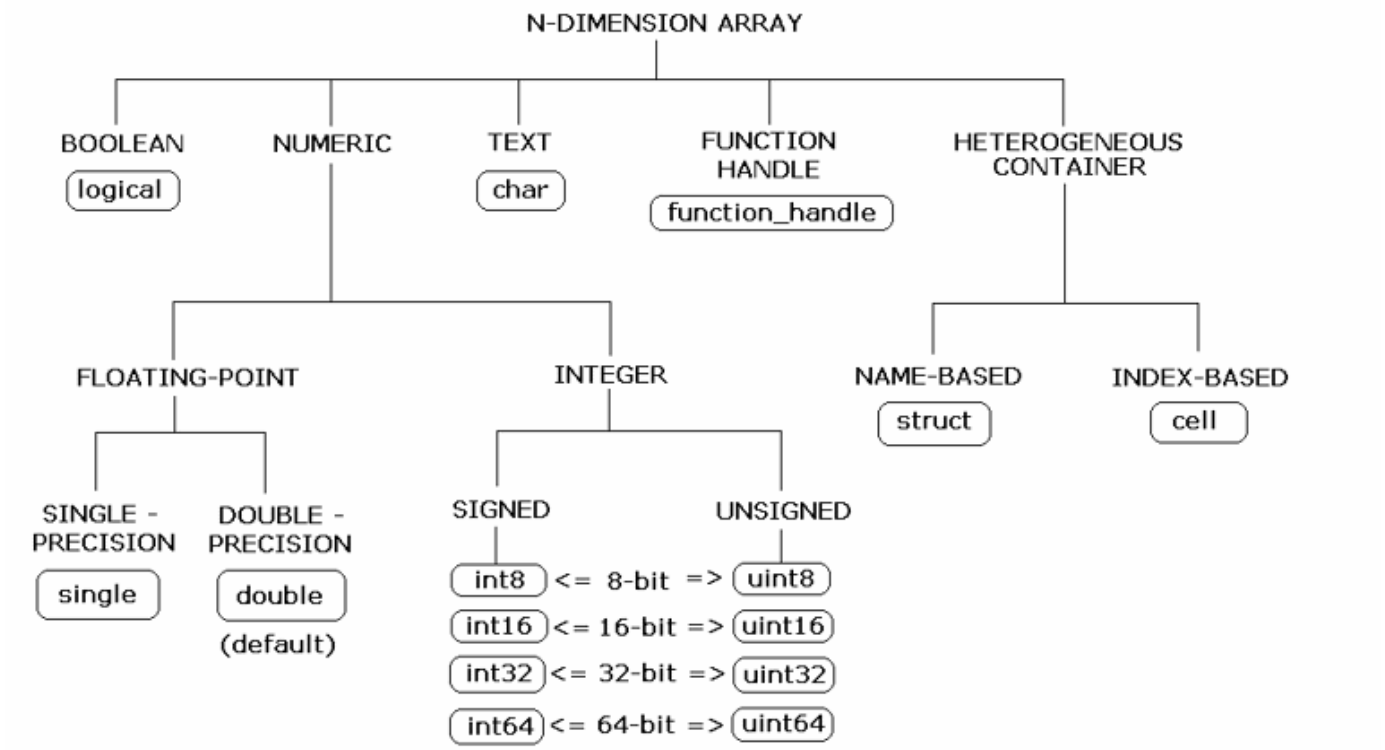
When the .NET client invokes a MATLAB function through a request and receives a result in the response, data conversion takes place between MATLAB types and C# types.

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 2-28.

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-2.

Scalar Numeric Type Coercion

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

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

MATLAB to .NET Numeric Type Compatibility describes the type compatibility for scalar numeric coercion.

MATLAB to .NET Numeric Type Compatibility

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

Dimension Coercion

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

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

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

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

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

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

Padding

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

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

How Your C# Method Return Type is Padded

MATLAB Function	C# Method Signature	When Dimensions in MATLAB are:	And Dimensions in C# are:
`function a = fooa = 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 = fooa = 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();
```

A

Data Conversion Rules

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` (character arrays 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.

B

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 on page B-4> 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 on page B-4` representing MATLAB stack trace. An unmodifiable copy of this list is made

Public Instance Properties

MATLABStackTrace

Returns list of `MATLABStackFrame on page B-4`

Gets MATLAB stack with `0` or more `MATLABStackFrame on page B-4`.

Each stack frame provides information about MATLAB file, function name, and line number. The output list of `MATLABStackFrame on page B-4` 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 on page B-4`

MATLABStackFrame

About MATLABStackFrame

Use `MATLABStackFrame` on page B-4 to return an element in MATLAB stack trace obtained using `MATLABException` on page B-2.

`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 on page B-2`

MWClient

About MWClient

Interface of `MWHttpClient` on page B-8, 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 deployable archive hosted by the server. The deployable archive is identified by a URL.

The methods in returned proxy object match the names of MATLAB functions in the deployable 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 deployable archive, with the form of `http://`*`localhost`*`:`*`port_number`*`/`*`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` on page B-8

MWHttpClient

About MWHttpClient

Implements `MWClient` on page B-6 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 deployable archive hosted by the server. The deployable archive is made available to the client as a URL.

A server can host multiple deployable archives since each deployable archive has a unique URL.

In order to establish client-server communication, the following is required:

- URL to the deployable archive in the form: `http://`*`localhost`*`:`*`port_number`*`/`*`archive_name_without_extension`*
- Names of MATLAB functions exported by the deployable 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 deployable archive. 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 deployable 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 deployable archive hosted by the server. The deployable archive is identified by a URL.

The methods in returned proxy object match the names of MATLAB functions in the deployable 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 deployable archive, with the form of `http://`*`localhost`*`:`*`port_number`*`/`*`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` on page B-6

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`