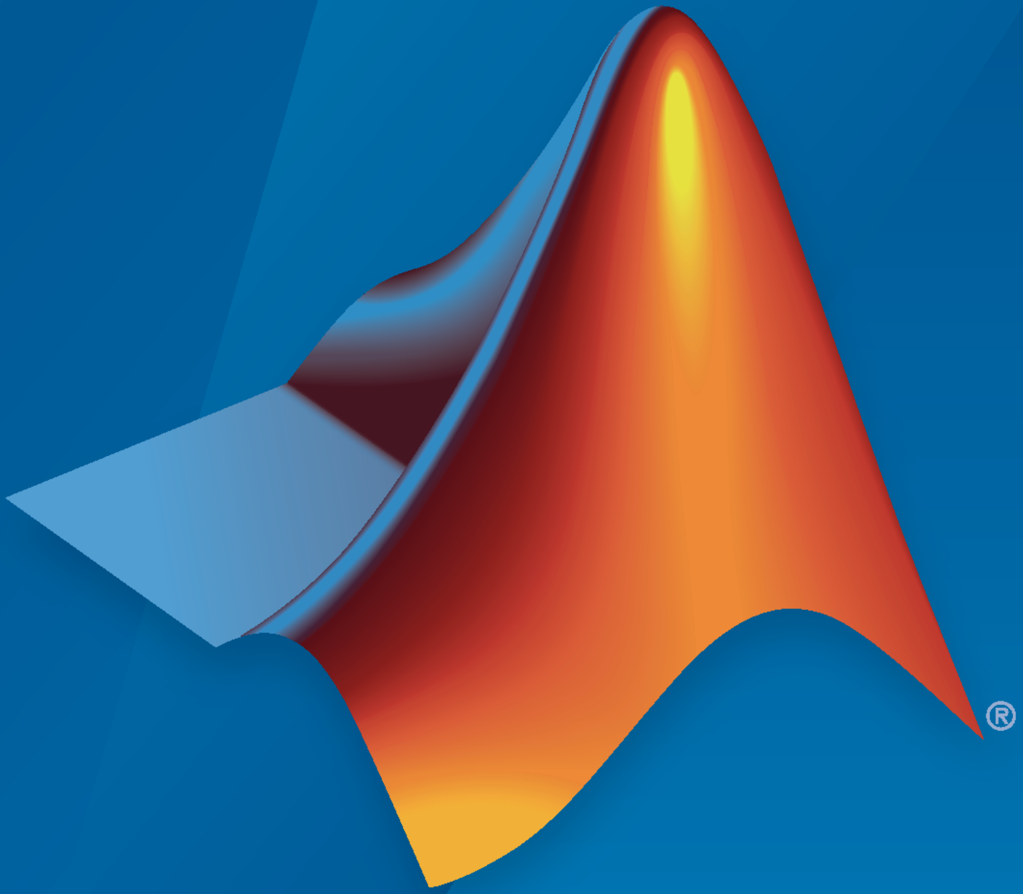
MATLAB® Production Server™

.NET Programming Guide



MATLAB®

R2022a

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)
September 2020	Online only	Revised for Version 4.4 (Release R2020b)
March 2021	Online only	Revised for Version 4.5 (Release R2021a)
September 2021	Online only	Revised for Version 4.6 (Release R2021b)
March 2022	Online only	Revised for Version 5.0 (Release R2022a)

Contents

Client Programming

Create a .NET MATLAB Production Server Client

To create a MATLAB Production Server client:

1. Obtain the client runtime files located in *`$MPS_INSTALL`*`/client/dotnet`. You can also download the client runtime files from MATLAB Production Server Client Libraries.
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, or the MATLAB Production Server "RESTful API for MATLAB Function Execution".

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

     For more information, 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.

     For more information, see "Invoke MATLAB Functions Dynamically" on page 2-7.
   - The .NET client RESTful API uses protobuf for data serialization. For an example, see "Asynchronous RESTful Requests Using Protocol Buffers in .NET Client" on page 2-35.
5. If your client uses a proxy, write .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.
   b. Declare and throw exceptions as required.
   c. Free system resources using the `close` method of `MWClient`, after making needed calls to your application.
6. If your client uses the RESTful API, to use protobuf when making a request to the server, set the HTTP `Content-Type` request header to `application/x-google-protobuf` in the client code. The .NET client library provides helper classes to internally create protobuf messages based on a proto format and returns the corresponding byte array. Use this byte array in the HTTP request body. The .NET client library provides methods and classes to deserialize the protobuf responses.

See Also

More About

- "Create a C# Client" on page 1-3
- "Asynchronous RESTful Requests Using Protocol Buffers in .NET Client" on page 2-35
- "Synchronous RESTful Requests Using Protocol Buffers in .NET Client" on page 2-42

Create a C# Client

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

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

Create Microsoft Visual Studio Project

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

Create Reference to Client Runtime Library

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

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

   In an on-premises MATLAB Production Server installation, the library is located in *`$MPS_INSTALL`*`\client\dotnet`, where *`$MPS_INSTALL`* is the location in which MATLAB Production Server is installed. Select the `MathWorks.MATLAB.ProductionServer.Client.dll` file.

   The client library is also available for download at `https://www.mathworks.com/products/matlab-production-server/client-libraries.html`.
3. Click **OK**. Your Microsoft Visual Studio project now references the `MathWorks.MATLAB.ProductionServer.Client.dll`.

Deploy MATLAB Function to Server

Write a MATLAB function `mymagic` that uses the `magic` function to create a magic square, package `mymagic` into a deployable archive called `mymagic_deployed`, then deploy it to a server. The function `mymagic` takes a single `int` input and returns a magic square as a 2-D `double` array. The example assumes that the server instance is running at `http://localhost:9910`.

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

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

Design .NET Interface in C#

Invoke the deployed MATLAB function `mymagic` from a .NET client through a .NET interface. Design a C# interface `Magic` to match the MATLAB function `mymagic`.

- The .NET interface has the same number of inputs and outputs as the MATLAB function.
- Since you are deploying one MATLAB function on the server, you define one corresponding .NET method in your C# code.
- Both the MATLAB function and the .NET interface process the same data types—input type `int` and output type 2-D `double`.
- In your C# client program, use the interface `Magic` to specify the type of the proxy object reference in the `CreateProxy` method. The `CreateProxy` method requires the URL to the deployable archive that contains the `mymagic` function (`http://localhost:9910/mymagic_deployed`) as an input argument.

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

Write, Build, and Run .NET Application

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

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

namespace Magic
{
    public class MagicClass
    {

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

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

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

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

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

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

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

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

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

See Also

More About

- “Create a .NET MATLAB Production Server Client” on page 1-2
- “Configure the Client-Server Connection” on page 2-5
- “Synchronous RESTful Requests Using Protocol Buffers in .NET Client” on page 2-42

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. For a list of supported software, including IDEs and Microsoft .NET Framework, see Supported and Compatible Compilers.
2. Verify that your application is deployed to a running server instance.

Handling Exceptions

The following table lists errors and the corresponding method to use to declare exceptions.

Error	Method	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 1-6

Where to Find the API Documentation

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

See Also

Related Examples

- "Unsupported MATLAB Data Types for Client and Server Marshaling" on page 1-6
- "Create a C# Client" on page 1-3

Prepare Your Microsoft Visual Studio Environment

Before you begin writing your .NET client program using the MATLAB Production Server .NET client library, complete the following steps to prepare your development environment.

Create Microsoft Visual Studio Project

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

Add Reference to .NET Client Library

To use the classes and methods defined in the MATLAB Production Server .NET client library, create a reference to the library in your Microsoft Visual Studio project.

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

   In an on-premises MATLAB Production Server installation, the library is located in *`$MPS_INSTALL`*`\client\dotnet`, where *`$MPS_INSTALL`* is the location in which MATLAB Production Server is installed. Select the `MathWorks.MATLAB.ProductionServer.Client.dll` file.

   The client library is also available for download at `https://www.mathworks.com/products/matlab-production-server/client-libraries.html`.
3. Click **OK**. Your Microsoft Visual Studio project now references the `MathWorks.MATLAB.ProductionServer.Client.dll`.

See Also

More About

- "Create a .NET MATLAB Production Server Client" on page 1-2
- "Create a C# Client" on page 1-3

Configure the Client-Server Connection

In this section...
“Create Connection with Default Configuration” on page 2-5
“Create Connection with Custom Configuration” on page 2-5
“Implementing Custom Connection Configuration” on page 2-6

You configure the client-server connection using an object that implements the `MWHttpClientConfig` interface. The 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 Connection with 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 Connection with 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)`

The following 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 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 second 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; }
  }
}
```

See Also

More About

- “Create a C# Client” on page 1-3

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 Reflection-Based Proxy

A reflection-based proxy implements the `MWInvokable` interface and provides methods that allow you to 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 MATLAB Function with 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 2-21.

For more information about the other data marshaling rules, see “Data Conversion with C# and MATLAB Types” on page 2-29.

See Also

Related Examples

- “Create a C# Client” on page 1-3
- “Create a .NET MATLAB Production Server Client” on page 1-2

Execute MATLAB Functions Using HTTPS

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 instance is configured to use HTTPS. For more information, see “Enable HTTPS”.
2. Configure the client environment for using SSL.
3. Create the program proxy using the HTTPS URL of the deployed application. For more information about writing a client program using a proxy, see “Create a C# Client” on page 1-3.

Configure Client Environment for SSL

Before your client application can send HTTPS requests to a server instance, the root SSL certificate of the server must be present in the Windows® Trusted Root Certification Authorities certificate store on the client machine. If the server uses a self-signed SSL certificate or if the root certificate of the server signed by a certificate authority (CA) is not present in the Windows certificate store, obtain the server certificate from the MATLAB Production Server administrator or export the certificate using a browser, then add it to the Windows certificate store.

Export and Save SSL Certificate

You can use any browser to save the server certificate on the client machine. The procedure to save the certificate using Google Chrome™ follows.

1. Navigate to the server instance URL `https://`*`server FQDN`*`:`*`port`*`/api/health` using Google Chrome.
2. In the Google Chrome address bar, click the padlock icon or the warning icon, depending on whether the server instance uses a CA-signed SSL certificate or a self-signed SSL certificate.
3. Click **Certificate** > **Details** > **Copy to File**. Doing so opens a wizard that lets you export the SSL certificate. Click **Next**.
4. Select the format to export the certificate and click **Next**.
5. Specify the location and file name to export the certificate, then click **Next**.
6. Click **Finish** to complete exporting the certificate.

Add Certificate to Windows Certificate Store

You can use a certificate management tool or Microsoft Management Console (MMC) to add the server certificate to the Windows certificate store. The procedure to add the certificate using MMC follows.

1. Open MMC from your Windows machine.
2. Click **File** > **Add/Remove Snap-in**. Doing so opens the **Add or Remove Snap-ins** window.
3. In the **Add or Remove Snap-ins** window:
   a. Click **Certificates** from the left pane, then click **Add**.
   b. Select **Computer account**, then click **Finish**. Doing so adds **Certificates(Local Computer)** to the right pane.
   c. Click **OK**. Doing so takes you to the home window.

4. In the left pane of the home window, under **Console Root**, double click **Certificates(Local Computer)**. Doing so opens all the certificate folders located in the local machine.
5. Select **Trusted Root Certification Authorities > More Actions > All Tasks > Import**. Doing so opens the Certificate Import Wizard.
6. Click **Next**, then select the location of your server certificate.
7. Click **Next** to import the certificate in the Trusted Root Certification Authorities certificate store.

Establish Secure Proxy Connection Without Client Authentication

After your client machine is configured to use the server certificate, you can write your client program to create a secure proxy connection with the server using the following code:

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

Doing so creates a secure proxy connection with the server instance running at `https://`*`server FQDN`*`:`*`port`* to communicate with the deployed application `myApplication`. The connection uses the `MWHttpClient` constructor and the proxy object reference `sslProxy`.

`sslProxy` checks the certificate stores of the client machine to perform the HTTPS server authentication. If the server requests client authentication, the HTTPS handshake fails because the client does not have a certificate.

Establish Secure Proxy Connection Using Client Authentication

Before a .NET client can communicate with a server instance that requires client authentication, you must create a client certificate bundle on the client machine and save the client certificate on the server instance.

Create and Merge Client Certificate

1. On the client machine, generate a self-signed SSL certificate and private key, or obtain a CA-signed SSL certificate and private key.

   To generate a self-signed SSL certificate, you can use the `openssl` command as follows:

   ```
   openssl req -x509 -nodes -newkey rsa:4096 -keyout client_key.pem -out client_cert.pem -days 3
   ```

   The command generates a self-signed certificate `client_cert.pem` with a private key `client_key.pem`. The certificate is valid for 365 days. For more information, see OpenSSL.

   The MATLAB Production Server administrator must save the client certificate `client_cert.pem` on the server instance and set the x509-ca-file-store in the server configuration file `main_config`. For information on configuring the server for client authentication, see “Configure Client Authentication”.
2. On the client machine, merge the client certificate and private key into a PKCS#12 (PFX) file by using the following command:

   ```
   openssl pkcs12 -export -in client_cert.pem -inkey client_key.pem -out client_certificate.pfx
   ```

Write .NET Client Program

1 Implement the `MWSSLConfig` interface.

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

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

2 Create a secure proxy connection to the server.

Create a secure proxy connection with a server instance using the `MWHttpClient` constructor. The `MWHttpClient` constructor takes as an argument an instance of your `MWSSLConfig` implementation. Create an interface-based proxy object reference with the HTTPS URL for the desired application using the `createProxy` method.

```
MWClient client = new MWHttpClient(new ClientSSLConfig());
Uri secureUri = new Uri("https://<server FQDN>:9920/myApplication")
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.

Handle Exceptions

Override Certificate Check

If the self-signed certificate or the root CA certificate of the server is not present in the Windows Trusted Root Certification Authorities certificate store on the client machine, and there is no mismatch between the host name of the HTTPS URL for MATLAB function execution and the common name (CN) of the SSL certificate of the server, then running your client program results in the following exception:

```
No response received in WebException with status : TrustFailure
```

Use one of the following options to handle this exception:

- Add the SSL certificate of the server to the Windows Trusted Root Certification Authorities certificate store on the client machine. For more information, see “Configure Client Environment for SSL” on page 2-10.
- Override the certificate check and accept the untrusted certificate using the following code:

  ```
  ServicePointManager.ServerCertificateValidationCallback += (sender, certificate, chain, sslPol
  ```

  This option is not recommended for a production environment, as it overrides all certificate checks.

  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.

Disable Host Name Verification

If there is a mismatch between the host name of the HTTPS URL for MATLAB function execution and the CN of the SSL certificate on the server, you can override the certificate check to disable host name verification using the following code in your client program:

```
ServicePointManager.ServerCertificateValidationCallback = delegate (
Object obj, X509Certificate certificate, X509Chain chain, SslPolicyErrors errors)
{
  if (errors.ToString().Equals("RemoteCertificateNameMismatch"))
  {
    return (true);
  }
  return (false);
};
```

A MATLAB Production Server deployment on Azure® uses a self-signed SSL certificate by default. Replacing the self-signed certificate with a CA-signed certificate is recommended. However, if you want to use the self-signed certificate and send HTTPS requests to the server, client programs must disable host name verification to avoid encountering an exception caused by a failure in host name verification. The verification fails due to a mismatch between the host names in the HTTPS URL for MATLAB function execution and the common name (CN) of the self-signed certificate. The host name for the MATLAB execution endpoint has the value `<uniqueID>.<location>.cloudapp.azure.com`, but the CN has the value `azure.com`. For information about MATLAB Production Server on Azure, see "Azure Deployment for MATLAB Production Server (BYOL)" and "Azure Deployment for MATLAB Production Server (PAYG)".

Implement Advanced Authentication Features

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

- Disable SSL protocols to protect against the POODLE exploit.

  ```
  System.Net.ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls | SecurityProtocolT
  ```
- Perform alternate host name verification to authenticate servers when the host name in the server URL does not match the host name in the SSL certificate.
- Ensure that the client shares data only with specific servers.

Sample Code

Sample client program for communicating with a server using HTTPS follows.

MagicProxy.cs

```
using System;
using System.Net;
using System.Security.Cryptography.X509Certificates;
using MathWorks.MATLAB.ProductionServer.Client;
using System.Net.Security;
```

```
namespace MagicSquareExample
{
    class MySSLConfig : MWSSLConfig
    {
        public X509CertificateCollection ClientCertificates
        {
            get
            {
                return new X509CertificateCollection();
            }
        }
    }

    class MagicProxy
    {
        static void Main(string[] args)
        {
            MWClient client = new MWHttpClient();

            //Disable hostname verification
            ServicePointManager.ServerCertificateValidationCallback = delegate (
            Object obj, X509Certificate certificate, X509Chain chain,
            SslPolicyErrors errors)
            {
                if (errors.ToString().Equals("RemoteCertificateNameMismatch"))
                {
                    return (true);
                }
                return (false);
            };

            try
            {
                MWInvokable invokable = client.CreateComponentProxy(new Uri("https://localhost:9
                double[,] magic = invokable.Invoke<Double[,]>("mymagic", 5);

                PrintMagic(magic);
            }
            catch (MATLABException ex)
            {
                Console.WriteLine("MATLAB error thrown : ");
                Console.WriteLine(ex.MATLABIdentifier);
                Console.WriteLine(ex.MATLABStackTraceString);
            }

            finally
            {
                if (client != null)
                {
                    client.Dispose();
                }
            }

            Console.ReadKey();
        }

        static void PrintMagic(double[,] magic)
        {
            int numDims = magic.Rank;
            int[] dims = new int[numDims];

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

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

See Also

More About

- “Configure the Client-Server Connection” on page 2-5
- “Enable HTTPS”
- “Create a C# Client” on page 1-3

External Websites

- MSDN documentation
- .Net `ServicePointManager` documentation

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 2-29 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 2-29 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 2-16), 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 2-18, 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 2-29 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 `structs` and classes.

In MATLAB, create a structure that contains `name`, `score`, and `grade` to store information for a student.

```
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+'
```

You can create a multidimensional structure array by inserting additional elements in the structure `S`. The following code creates a structure array of dimensions `(1,3)`.

```
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 student score. 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 2-21 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 the 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. For more information about this custom attribute, see the .NET 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 2-21, 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 2-26
- "Interface Attributes" on page 2-26
- "Fields and Property Attributes" on page 2-27

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.

See Also

More About

- “Create a .NET MATLAB Production Server Client” on page 1-2
- “Create a C# Client” on page 1-3
- “Struct Support for RESTful Requests Using Protocol Buffers in .NET Client” on page 2-47

Data Conversion with C# and MATLAB Types

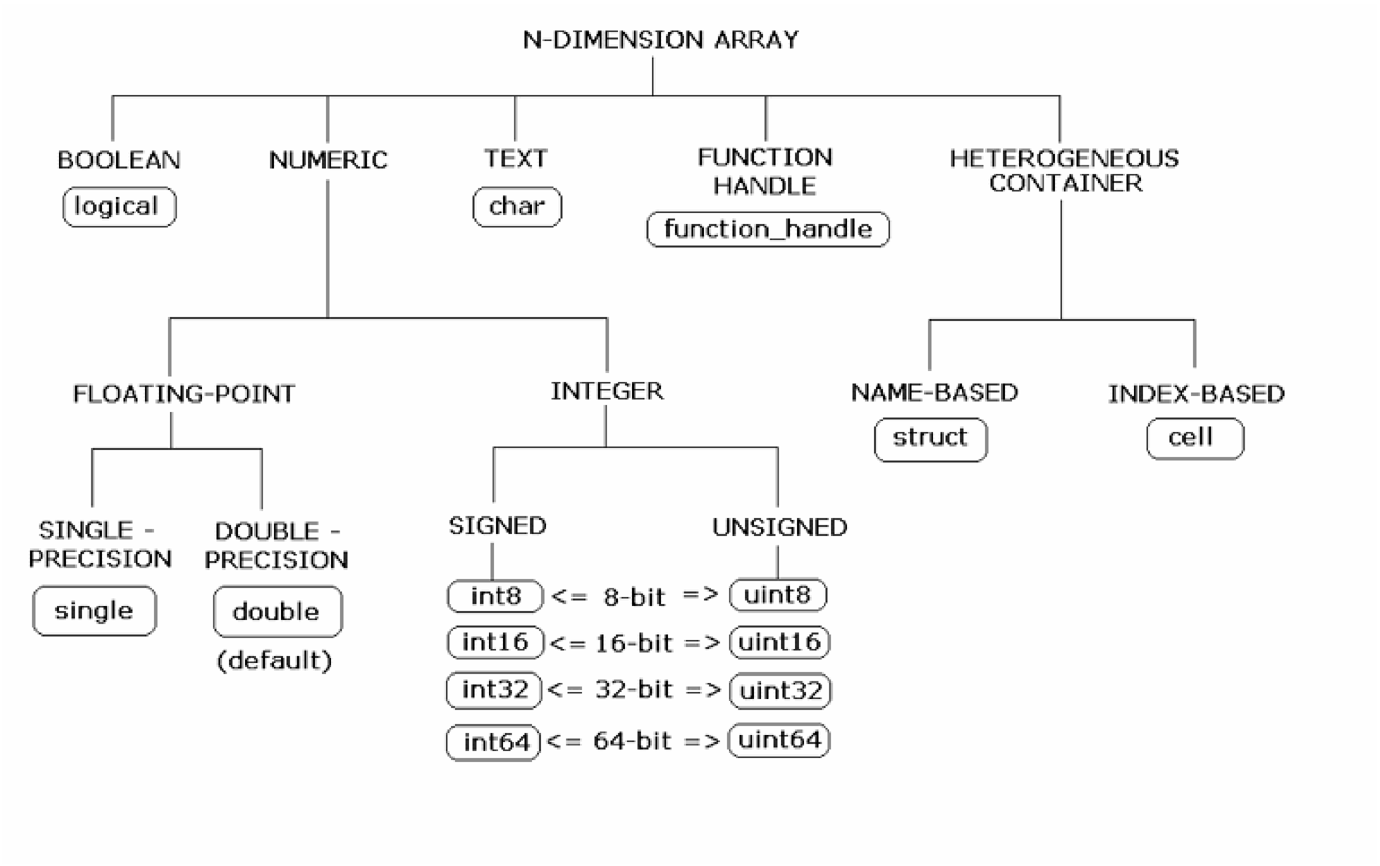
When the .NET client invokes a MATLAB function through a server request and receives a result in the response, data conversion takes place between MATLAB data types and C# data types.

Working with MATLAB Data Types

There are several 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.

Note Function Handles are not supported by MATLAB Production Server.

The fundamental MATLAB classes are circled in the following diagram.



Each MATLAB data type has a specific equivalent in C#. For a mapping of these one-to-one relationships, see “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 following table describes the type compatibility for scalar numeric coercion.

MATLAB to .NET Numeric Type Compatibility

MATLAB Type	.NET Types	C# Type
uint8	System.Byte, System.Int16, System.UInt16, System.Int32, System.UInt32, System.Int64, System.UInt64, System.Single, System.Double	byte
int8	System.SByte, System.Int16, System.Int32, System.Int64, System.Single, System.Double	sbyte
uint16	System.UInt16, System.Int32, System.UInt32, System.Int64, System.UInt64, System.Single, System.Double	ushort
int16	System.Int16, System.Int32, System.Int64, System.Single, System.Double	short
uint32	System.UInt32, System.Int64, System.UInt64, System.Single, System.Double	uint
int32	System.Int32, System.Int64, System.Single, System.Double	int
uint64	System.UInt64, System.Single, System.Double	ulong
int64	System.Int64, System.Single, System.Double	long
single	System.Single, System.Double	float

MATLAB Type	.NET Types	C# Type
`double`	`System.Double`, `System.SByte`, `System.Byte`, `System.Int16`, `System.UInt16`, `System.Int32`, `System.UInt32`, `System.Int64`, `System.UInt64`, `System.Single`	`double`

Dimension Coercion

In MATLAB, dimensionality is an attribute of the fundamental data types and does not add to the number of data types as it does in .NET. For example, 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. For example, if your MATLAB function returns a three-dimensional double array, your C# function declares its return value as a C# three-dimensional `double` array

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

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

The .NET client API performs the coercion 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 the return type of a C# method has a greater number of dimensions than that of a MATLAB method, the dimensions of the MATLAB return type are padded with ones (1s) to match the required number of output dimensions in C#.

The following table provides examples of the padding. For example, when your MATLAB function returns a two-dimensional matrix but your C# function declares a four-dimensional array return value, the MATLAB matrix becomes a four-dimensional array with two trailing singular dimensions.

C# Method Return Type Padding

MATLAB Function	C# Method Signature	Dimensions in MATLAB	Dimensions in C#

`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 return type of a C# method has fewer dimensions than the dimensions of the return type of a MATLAB method, the dimensions of the MATLAB return type 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 (`1`s) only.

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

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

The following tables provide examples of truncation.

How MATLAB Truncates Your C# Method Return Type

MATLAB Function	C# Method Signature	Dimensions in MATLAB	Dimensions in C#

`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 example, you can return an empty array from `function a = foo` by assigning `a` to any of the following values:

```
 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();
```

See Also

More About

- "Data Conversion for .NET and MATLAB Types" on page 2-3
- "Unsupported MATLAB Data Types for Client and Server Marshaling" on page 1-6

Asynchronous RESTful Requests Using Protocol Buffers in .NET Client

This example shows how to make asynchronous RESTful requests using the .NET client API, MATLAB Production Server "RESTful API for MATLAB Function Execution", and protocol buffers (protobuf). The example provides and explains a sample C# client, `MagicAsync.cs`, for evaluating a MATLAB function deployed on the server.

To use protobuf when making a request to the server, set the HTTP `Content-Type` request header to `application/x-google-protobuf` in the client code. The `MathWorks.MATLAB.ProductionServer.Client.REST` namespace in the .NET client library provides helper classes to internally create protobuf messages based on a proto format and returns the corresponding byte array. Use this byte array in the HTTP request body. The .NET client library provides methods and classes to deserialize the protobuf responses.

When sending a POST Asynchronous Request to the server, the client must set the HTTP request `mode` to `async` in the query parameter. The request state must be either `READY` or `ERROR` before you can retrieve the request result. For more information about asynchronous request execution on MATLAB Production Server, see "Asynchronous Execution".

To use the .NET client API, you must add a reference to the `MathWorks.MATLAB.ProductionServer.Client.dll` file in your C# project. For more information on preparing your Microsoft Visual Studio environment for your project, see "Prepare Your Microsoft Visual Studio Environment" on page 2-4.

In an on-premises MATLAB Production Server installation, the client APIs are located in *`$MPS_INSTALL`*`/client`, where *`$MPS_INSTALL`* is the MATLAB Production Server installation location. The client APIs are also available for download at MATLAB Production Server Client Libraries. The Java® client API is also hosted in a Maven™ repository at https://mvnrepository.com/artifact/com.mathworks.prodserver/mps_java_client.

Deploy MATLAB Function to Server

Write a MATLAB function `mymagic` that uses the `magic` function to create a magic square, then deploy it on the server. The function `mymagic` takes a single `int32` input and returns a magic square as a 2-D `double` array. The example assumes that the server instance is running at `http://localhost:9910`.

For information on how to deploy, see "Create Deployable Archive for MATLAB Production Server".

```
function m = mymagic(in)

  m = magic(in);
end
```

Make Asynchronous Request to Server

In the C# client code, use the POST Asynchronous Request to make the initial request to the server. For more information about asynchronous request execution in MATLAB Production Server, see "Asynchronous Execution".

1 Create an HTTP request body containing the protocol buffer message.

Use the `Create(arg1, arg2, arg3)` method defined in the `MATLABParams` class of the MATLAB Production Server .NET client API to build the protocol buffer message. The `Create` method takes as input the expected number of output arguments for the deployed function, the expected output type, and an array of objects representing the inputs to the deployed function. Since the deployed `mymagic` function returns a single 2-D array, set `arg1` to `1` and `arg2` to `new List<Type> { typeof(double[,]) }`. Specify an integer value for `arg3`, which is the input to the `mymagic` function.

Create a POST Asynchronous Request using the .NET `WebRequest.Create` method. For more information on the .NET `WebRequest` class, see Microsoft documentation.

```
MATLABParams mlParams = MATLABParams.Create(1, new List<Type> { typeof(double[,]) }, 2);
```

2 Send the request to the server.

Send a POST Asynchronous Request to the server where the request body consists of the protobuf message (`mlParams`). The request URL comprises the address of the server instance (`http://localhost:9910`), the name of the deployed archive (`mymagic`), and the name of the MATLAB function to evaluate (`mymagic`). Set the HTTP request `mode` to `async` in the query parameter. Set the HTTP `Content-Type` request header to `application/x-google-protobuf`, as the API returns a byte array of protocol buffer messages.

Send the request to the server using the .NET `WebRequest.getResponse` method. For more information on the .NET `WebRequest` class, see Microsoft documentation.

```
String mpsBaseUrl = "http://localhost:9910";
var response = MakeHTTPRequest(mpsBaseUrl + "/mymagic/mymagic?mode=async", "POST", mlParams);
Console.WriteLine("The HTTP status code of the request is " + response.StatusCode + ".\n\n");
```

The example uses a helper method `MakeHTTPRequest` to send the protobuf message to the server. This method takes as input an HTTP URL, an HTTP method (GET or POST), and a `MATLABParams` object, and returns the server response.

```
static HttpWebResponse MakeHTTPRequest(String url, String requestType, MATLABParams mlParams)
{
    var httpRequest = (HttpWebRequest)WebRequest.Create(url);
    httpRequest.Method = requestType;
    httpRequest.ContentType = "application/x-google-protobuf";
    if (requestType.Equals("POST"))
    {
        mlParams.WriteTo(httpRequest.GetRequestStream());
    }
    return (HttpWebResponse)httpRequest.GetResponse();
}
```

3 Receive and interpret the server response.

On successful execution of the HTTP requests, the server responds with a protocol buffer message. Parse the protocol buffer message using methods from the `MATLABRequestHandle` class to get details such as the state of the request, the request URL, and the last modified sequence value of the request.

```
MATLABRequestHandle matlabRequestHandle = MATLABRequestHandle.Create(response.GetResponseStre
Console.WriteLine("The response body of the initial POST request contains the following value
Console.WriteLine(matlabRequestHandle.ToString() + ".\n\n");
```

Get State Information of Request

1 Make a request to get the request state information.

   Make a request to the GET State Information RESTful API. In the request URL, set the query parameter `format` to `protobuf` so that the server returns the output in protocol buffer format. You can get the result of an asynchronous request only after the state of the request has changed to `READY` or `ERROR`.

2 Parse the response.

   Parse the response using methods defined in the `MATLABRequest` class to get the state of the request.

The example code makes a request to the GET State Information API every second to look for a request state change to either `READY_STATE` or `ERROR_STATE`.

```
 MATLABRequest mlRequestStatus;
 do
 {
    var statusRequestResponse = MakeHTTPRequest(mpsBaseUrl + matlabRequestHandle.RequestURL + "/i
    mlRequestStatus = MATLABRequest.Create(statusRequestResponse.GetResponseStream());
    Console.WriteLine("State: " + mlRequestStatus.State);
    Thread.Sleep(1000);
}while (mlRequestStatus.State < MATLABRequestState.READY_STATE);
```

In asynchronous mode, a client is able to post multiple requests to the server. To get the state information of each POST request, you must make a corresponding request to the GET State Information RESTful API.

Retrieve Results of Request

1 Make a request to fetch the response.

   Use the GET Result of Request RESTful API to fetch the request results after the request state has changed to `READY` or `ERROR`. In the request URL, set the query parameter `format` to `protobuf`, so that the server returns the output in protocol buffer format.

```
response = MakeHTTPRequest(mpsBaseUrl + matlabRequestHandle.RequestURL + "/result?format=prot
```

2 Parse the response.

   If the request state is `READY`, use the methods defined in the `MATLABResult` class to parse the response. To create a `MATLABResult` object, use the `Create` method, and pass as inputs the `MATLABParams` `mlParams` object and the response body of the request to GET Result of Request API.

   If an error occurs when the deployed MATLAB function executes, the call to the `Result` method throws a `MATLABException` that contains the error message from MATLAB.

   If the request state is `ERROR`, use the `HTTPErrorInfo` class instead of the `MATLABResult` class to parse the response. Use the methods defined in the `HTTPErrorInfo` class to get information about the error.

```
if (mlRequestStatus.State == MATLABRequestState.READY_STATE)
{
    MATLABResult mlResult;
    try
    {
         mlResult = MATLABResult.Create(mlParams, response.GetResponseStream());
         double[,] result = mlResult.Result<double[,]>();
         Console.WriteLine("Printing the 2-D array...\n");
         PrintMagic(result);
    }
    catch (MATLABException e)
    {
        Console.WriteLine(e.Message);
    }
}
else if (mlRequestStatus.State == MATLABRequestState.ERROR_STATE)
{
    HTTPErrorInfo httpErrorInfo = HTTPErrorInfo.Create(response.GetResponseStream());
    Console.WriteLine("Error:");
    Console.WriteLine(httpErrorInfo.HttpErrorCode);
    Console.WriteLine(httpErrorInfo.HttpErrorMessage);
}
```

3 Display the results.

The example uses a helper method `PrintMagic` that takes as input the result that is parsed from the response body of the GET Result of Request API call and prints the corresponding 2-D magic square array.

```
static void PrintMagic(double[,] magic)
{
    int numDims = magic.Rank;
    int[] dims = new int[numDims];

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

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

Running the C# application generates the following output.

```
Printing the 2-D array...

1,3
4,2
```

Sample code for the `MagicAsync.cs` C# client follows.

Code:

MagicAsync.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.IO;
using MathWorks.MATLAB.ProductionServer.Client;
using MathWorks.MATLAB.ProductionServer.Client.REST;
using System.Threading;

namespace MagicSquareExample
{
    class MagicAsync
    {
        static void Main(string[] args)
        {
            //Async request using RESTful API and protobuf

            //URL of the MATLAB Production Server
            String mpsBaseUrl = "http://localhost:9910";
            MATLABParams mlParams = MATLABParams.Create(1, new List<Type> { typeof(double[,]) },

            Console.WriteLine("Sending initial HTTP POST request to the URL " + mpsBaseUrl + "/my
            var response = MakeHTTPRequest(mpsBaseUrl + "/mymagic/mymagic?mode=async", "POST", m
            Console.WriteLine("The HTTP status code of the request is " + response.StatusCode + "

            // Parse the response body using the MATLABRequestHandle class
            // to retrieve the request URL, last modified sequence value and state of the request
            MATLABRequestHandle matlabRequestHandle = MATLABRequestHandle.Create(response.GetResp
            Console.WriteLine("The response body of initial POST request contains the following v
            Console.WriteLine(matlabRequestHandle.ToString() + ".\n\n");

            Console.WriteLine("Send GET status request every 1 second to look for state change of
            MATLABRequest mlRequestStatus;
            do
            {
                var statusRequestResponse = MakeHTTPRequest(mpsBaseUrl + matlabRequestHandle.Requ
                mlRequestStatus = MATLABRequest.Create(statusRequestResponse.GetResponseStream()
                Console.WriteLine("State: " + mlRequestStatus.State);
                Thread.Sleep(1000);
            } while (mlRequestStatus.State < MATLABRequestState.READY_STATE);
            //-----------
            // Once the state changes to READY_STATE or ERROR_STATE, make a GET call to /result.
            //use HTTPErrorInfo class, else use MATLABResult class to parse the response body.
            Console.Write("\n\n");
            Console.WriteLine("Now send GET request to fetch the result using the URL " + mpsBase
            response = MakeHTTPRequest(mpsBaseUrl + matlabRequestHandle.RequestURL + "/result?for

            if (mlRequestStatus.State == MATLABRequestState.READY_STATE)
            {
                // Parse the response body with the help of MATLABResult class. The Create functi
                // takes as input the MATLABParams object that was initially created.
                // If there is any error in MATLAB, call to Result<>() throws MATLABException whi
                // displayed in MATLAB.
                MATLABResult mlResult;
                try
                {
                    mlResult = MATLABResult.Create(mlParams, response.GetResponseStream());
                    double[,] result = mlResult.Result<double[,]>();
                    Console.WriteLine("Printing the 2-D array...\n");
                    PrintMagic(result);
                }
                catch (MATLABException e)
                {
                    Console.WriteLine(e.Message);
                }
            }
            else if (mlRequestStatus.State == MATLABRequestState.ERROR_STATE)
            {
                HTTPErrorInfo httpErrorInfo = HTTPErrorInfo.Create(response.GetResponseStream())
                Console.WriteLine("Error:");
                Console.WriteLine(httpErrorInfo.HttpErrorCode);
                Console.WriteLine(httpErrorInfo.HttpErrorMessage);
            }
            Console.ReadKey();
            //------------

        }

        static void PrintMagic(double[,] magic)
        {
            int numDims = magic.Rank;
            int[] dims = new int[numDims];

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

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

        //This function takes URL and HTTP request type and sends the HTTP request
        static HttpWebResponse MakeHTTPRequest(String url, String requestType, MATLABParams mlPa
        {
            var httpRequest = (HttpWebRequest)WebRequest.Create(url);
            httpRequest.Method = requestType;
            httpRequest.ContentType = "application/x-google-protobuf";
            if (requestType.Equals("POST"))
            {
                mlParams.WriteTo(httpRequest.GetRequestStream());
            }
            return (HttpWebResponse)httpRequest.GetResponse();
        }

    }
}
```

See Also

POST Asynchronous Request | GET Result of Request | GET State Information

More About

- “Create a .NET MATLAB Production Server Client” on page 1-2
- “Create a C# Client” on page 1-3
- POST Asynchronous Request
- “Synchronous RESTful Requests Using Protocol Buffers in .NET Client” on page 2-42
- “Struct Support for RESTful Requests Using Protocol Buffers in .NET Client” on page 2-47

Synchronous RESTful Requests Using Protocol Buffers in .NET Client

This example shows how to make synchronous RESTful requests using the .NET client API, MATLAB Production Server “RESTful API for MATLAB Function Execution”, and protocol buffers (protobuf). The example provides and explains a sample C# client, `MagicSync.cs`, for evaluating a MATLAB function deployed on the server.

To use protobuf when making a request to the server, set the HTTP `Content-Type` request header to `application/x-google-protobuf` in the client code. The `MathWorks.MATLAB.ProductionServer.Client.REST` namespace in the .NET client library provides helper classes to internally create protobuf messages based on a proto format and returns the corresponding byte array. Use this byte array in the HTTP request body. The .NET client library provides methods and classes to deserialize the protobuf responses.

In synchronous request execution, after a client posts a request, the server blocks all further requests until it has completed processing the original request. After processing is complete, the server automatically returns a response to the client. For more information, see “Synchronous Execution”.

To use the .NET client API, you must add a reference to the `MathWorks.MATLAB.ProductionServer.Client.dll` file in your C# project. For more information on preparing your Microsoft Visual Studio environment for your project, see “Prepare Your Microsoft Visual Studio Environment” on page 2-4.

In an on-premises MATLAB Production Server installation, the client APIs are located in *`$MPS_INSTALL`*`/client`, where *`$MPS_INSTALL`* is the MATLAB Production Server installation location. The client APIs are also available for download at MATLAB Production Server Client Libraries. The Java client API is also hosted in a Maven™ repository at https://mvnrepository.com/artifact/com.mathworks.prodserver/mps_java_client.

Deploy MATLAB Function to Server

Write a MATLAB function `mymagic` that uses the `magic` function to create a magic square, then deploy it on the server. The function `mymagic` takes a single `int32` input and returns a magic square as a 2-D `double` array. The example assumes that the server instance is running at `http://localhost:9910`.

For information on how to deploy, see “Create Deployable Archive for MATLAB Production Server”.

```
function m = mymagic(in)

  m = magic(in);
end
```

Make Synchronous Request to Server

In the C# client code, use the POST Synchronous Request to make the initial request to the server. For more information about synchronous request execution in MATLAB Production Server, see “Synchronous Execution”.

1 Create the request.

Create a request to the POST Synchronous Request RESTful API using the .NET `WebRequest.Create` method.

The request URL comprises the address of the server instance (`http://localhost:9910`), the name of the deployed archive (`mymagic`), and the name of the MATLAB function to evaluate (`mymagic`). Set the HTTP request method to `POST`. Set the HTTP `Content-Type` request header to `application/x-google-protobuf`, as the API returns a byte array of protocol buffer messages.

```
String mpsBaseUrl = "http://localhost:9910";
var firstRequest = (HttpWebRequest)WebRequest.Create(mpsBaseUrl + "/mymagic/mymagic");
firstRequest.Method = "POST";
firstRequest.ContentType = "application/x-google-protobuf";
```

2 Send the request to the server.

Send the request to the server using the .NET `WebRequest.getResponse` method.

Use the `Create(arg1, arg2, arg3)` method defined in the `MATLABParams` class present in `MathWorks.MATLAB.ProductionServer.Client.REST` namespace of the MATLAB Production Server .NET client API to build the protocol buffer message. The `Create` method takes as input the expected number of output arguments for the deployed function, the expected output type, and an array of objects representing the inputs to the deployed function. Since the deployed `mymagic` function returns a single 2-D array, set `arg1` to `1` and `arg2` to `new List<Type> { typeof(double[,]) }`. Specify an integer value for `arg3`, which is the input to the `mymagic` function.

```
MATLABParams mlParams = MATLABParams.Create(1, new List<Type> { typeof(double[,]) }, 2);
mlParams.WriteTo(firstRequest.GetRequestStream());
var response = (HttpWebResponse)firstRequest.GetResponse();
```

For more information on the `WebRequest` class, see Microsoft documentation.

Receive and Interpret Server Response

On successful execution of the POST Synchronous Request, the server responds with a protocol buffer message. Parse the protocol buffer message using methods from the `MATLABResult` class to get the result of the request. To create a `MATLABResult` object, use the `Create` method. Pass the `MATLABParams` `mlParams` object and the response body of the POST Synchronous Request to the `Create` method.

If an error occurs when the deployed MATLAB function executes, the call to the `Result` method throws a `MATLABException` that contains the error message from MATLAB.

```
MATLABResult mlResult;
mlResult = MATLABResult.Create(mlParams, response.GetResponseStream());
try
{
    double[,] result = mlResult.Result<double[,]>();
    Console.WriteLine("Printing the 2-D array...\n");
    PrintMagic(result);
}
catch (MATLABException e)
{
    Console.WriteLine(e.ToString());
}
```

The example uses a helper method `PrintMagic` that takes as input the response body of the POST Synchronous Request and prints the corresponding 2-D magic square array.

```
static void PrintMagic(double[,] magic)
{
    int numDims = magic.Rank;
    int[] dims = new int[numDims];

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

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

Running the C# application generates the following output.

```
Printing the 2-D array...

1,3
4,2
```

Sample code for the `MagicSync.cs` C# client follows.

Code:

MagicSync.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.IO;
using MathWorks.MATLAB.ProductionServer.Client;
using MathWorks.MATLAB.ProductionServer.Client.REST;
using System.Threading;

namespace MagicSquareExample
{
    class MagicSync
    {
        static void Main(string[] args)
        {
            //Sync request using RESTful API and protobuf

            // URL of the MATLAB Production Server.
            String mpsBaseUrl = "http://localhost:9910";

            Console.WriteLine("Sending the POST request to the URL: " + mpsBaseUrl + "/mymagic/my
            var firstRequest = (HttpWebRequest)WebRequest.Create(mpsBaseUrl + "/mymagic/mymagic")
            firstRequest.Method = "POST";

            //Set Content-Type HTTP header to protobuf.
            firstRequest.ContentType = "application/x-google-protobuf";

            //Use MATLABParams class to make POST request body.
            MATLABParams mlParams = MATLABParams.Create(1, new List<Type> { typeof(double[,]) },
            mlParams.WriteTo(firstRequest.GetRequestStream());
            var response = (HttpWebResponse)firstRequest.GetResponse();
            Console.WriteLine("The HTTP status code of the request is " + response.StatusCode + "

            //If there is a MATLAB error, call to Result<>() throws MATLABException which contai
            Console.WriteLine("Parsing the result from the response of the HTTP Request using MA
            MATLABResult mlResult;
            mlResult = MATLABResult.Create(mlParams, response.GetResponseStream());
            try
            {
                double[,] result = mlResult.Result<double[,]>();
                Console.WriteLine("Printing the 2-D array...\n");
                PrintMagic(result);
            }
            catch (MATLABException e)
            {
                Console.WriteLine(e.ToString());
            }
            Console.ReadKey();
        }

        static void PrintMagic(double[,] magic)
        {
            int numDims = magic.Rank;
            int[] dims = new int[numDims];

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

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

See Also

POST Synchronous Request

More About

- “Create a .NET MATLAB Production Server Client” on page 1-2
- “Create a C# Client” on page 1-3
- “Synchronous Execution”
- “Asynchronous RESTful Requests Using Protocol Buffers in .NET Client” on page 2-35
- “Struct Support for RESTful Requests Using Protocol Buffers in .NET Client” on page 2-47

Struct Support for RESTful Requests Using Protocol Buffers in .NET Client

This example shows how to send MATLAB structures (`struct` (MATLAB)) represented as arrays of .NET objects as input when you make a synchronous request using the .NET client API, MATLAB Production Server "RESTful API for MATLAB Function Execution", and protocol buffers (protobuf). The example provides and explains a sample Java client, `SortStudentsSyncREST.cs`, for evaluating a MATLAB function deployed on the server.

To use protobuf when making a request to the server, set the HTTP `Content-Type` request header to `application/x-google-protobuf` in the client code. The `MathWorks.MATLAB.ProductionServer.Client.REST` namespace in the .NET client library provides helper classes to internally create protobuf messages based on a proto format and returns the corresponding byte array. Use this byte array in the HTTP request body. The .NET client library provides methods and classes to deserialize the protobuf responses.

In synchronous request execution, after a client posts a request, the server blocks all further requests until it has completed processing the original request. After processing is complete, the server automatically returns a response to the client. For more information, see "Synchronous Execution".

To use the .NET client API, you must add a reference to the `MathWorks.MATLAB.ProductionServer.Client.dll` file in your C# project. For more information on preparing your Microsoft Visual Studio environment for your project, see "Prepare Your Microsoft Visual Studio Environment" on page 2-4.

In an on-premises MATLAB Production Server installation, the client APIs are located in *`$MPS_INSTALL`*`/client`, where *`$MPS_INSTALL`* is the MATLAB Production Server installation location. The client APIs are also available for download at MATLAB Production Server Client Libraries. The Java client API is also hosted in a Maven™ repository at https://mvnrepository.com/artifact/com.mathworks.prodserver/mps_java_client.

Deploy MATLAB function to server

Write a MATLAB function `sortstudents` that takes an array of structures containing student information as input and returns a sorted array of students based on their score. Student name, score and grade form the fields of the input structure. Deploy this function on the server. The example assumes that the server instance is running at `http://localhost:9910`.

For information on how to deploy, see "Create Deployable Archive for MATLAB Production Server".

Code for a sample MATLAB `struct` `S` and the MATLAB function `sortstudents` follows.

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

function sorted = sortstudents(unsorted)

scores = {unsorted.score};
scores = cell2mat(scores);
[s i] = sort(scores);
sorted = unsorted(i);
```

Create helper classes

Create .NET classes to marshal data to and from the MATLAB `struct`.

1 Create a .NET class `Student` with the same data members as the input structure.

```
class 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;
    }
}
```

2 Create a .NET class `StudentMarshaller` that extends the interface `MWDefaultMarshalingRules`. Since .NET does not natively support structs, extending the `MWDefaultMarshalingRules` interface lets you implement a new set of marshaling rules for the list of classes being marshaled, serialize .NET objects to structs and deserialize structs to .NET objects. Set the `StructTypes` property to return the *type* `Student`.

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

3 Create an array of type `Student` that you want to sort.

```
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 };
```

Make Synchronous Request to Server

In the C# client code, use the POST Synchronous Request to make the initial request to the server. For more information about synchronous request execution in MATLAB Production Server, see “Synchronous Execution”.

1 Create the request.

Create a POST Synchronous Request using the .NET `WebRequest.Create` method.

The request URL comprises the address of the server instance (`http://localhost:9910`), the name of the deployed archive (`sortstudents`), and the name of the MATLAB function to evaluate (`sortstudents`). Set the HTTP request method to `POST`. Set the HTTP `Content-Type` request header to `application/x-google-protobuf`, as the API returns a byte array of protocol buffer messages.

```
String mpsBaseUrl = "http://localhost:9910";
var firstRequest = (HttpWebRequest)WebRequest.Create(mpsBaseUrl + "/sortstudents/sortstudents
firstRequest.Method = "POST";
firstRequest.ContentType = "application/x-google-protobuf";
```

2 Send the request to the server.

Send the request to the server using the .NET `WebRequest.getResponse` method.

Use the `Create(arg1, arg2, arg3, arg4)` method defined in the `MATLABParams` class of the MATLAB Production Server .NET client API to build the protocol buffer message. The `Create` method takes as input the expected number of output arguments for the deployed function, the expected output type, an object of type `MWDefaultMarshalingRules`, and an array of objects representing the inputs to the deployed function. Since the deployed `sortstudents` function returns a single array of structs, set `arg1` to `1` and `arg2` to `new List<Type> { typeof(Student[]) }`. Set `arg3` to `new StudentMarshaler()`, and `arg4` to `new object[] { unsorted }`, which is the input to the `sortstudents` function.

```
MATLABParams mlParams = MATLABParams.Create(1, new List<Type> { typeof(Student[]) }, new Stud
mlParams.WriteTo(firstRequest.GetRequestStream());
var response = (HttpWebResponse)firstRequest.GetResponse();
```

For more information on the `WebRequest` class, see Microsoft documentation.

Receive and Interpret Server Response

On successful execution of the POST Synchronous Request, the server responds with a protocol buffer message. Parse the protocol buffer message using methods from the `MATLABResult` class to get the result of the request. To create a `MATLABResult` object, pass the `MATLABParams` `mlParams` object and the response body of the POST Synchronous Request to the `Create` method. Set the return type of the `MATLABResult` `mlResult` to `Student[]`.

If an error occurs when the deployed MATLAB function executes, the call to the `Result` method throws a `MATLABException` that contains the error message from MATLAB.

```
MATLABResult mlResult;
mlResult = MATLABResult.Create(mlParams, response.GetResponseStream());
try
{
    Student[] result = mlResult.Result<Student[]>();
    Console.WriteLine("Printing the sorted Student array...\n");
    PrintStudent(result);
}
catch (MATLABException e)
{
    Console.WriteLine(e.ToString());
}
```

The example uses a helper method `PrintStudent` that takes as input the response body of the POST Synchronous Request and prints the corresponding sorted list of students.

```
static void PrintStudent(Student[] students)
{
    foreach (Student s in students)
    {
        Console.WriteLine(s.ToString());
    }
}
```

Running the application generates the following output.

```
Printing the sorted Student array...

Ed Plum : B+ : 80
Mark Jones : A- : 85
Tony Miller : A : 90
```

Sample code for the `SortStudentsSyncREST.cs` C# client follows.

Code:

SortStudentsSyncREST.cs

```
using System;
using System.Net;
using MathWorks.MATLAB.ProductionServer.Client.REST;
using MathWorks.MATLAB.ProductionServer.Client;
using System.Collections.Generic;

namespace SortStudents
{
    class 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;
        }
    }

    class StudentMarshaler : MWDefaultMarshalingRules
    {
        public override IList<Type> StructTypes
        {
            get
            {
                return new List<Type> { typeof(Student) };
            }
        }
    }
    public class SortStudentsSyncREST
    {
        static void Main(String[] args)
        {
            //Struct example for the new API
            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 };

            // URL of the MATLAB Production Server.
            String mpsBaseUrl = "http://localhost:9910";

            Console.WriteLine("Sending the POST request to the URL: " + mpsBaseUrl + "/sortstuder
            var firstRequest = (HttpWebRequest)WebRequest.Create(mpsBaseUrl + "/sortstudents/sor
            firstRequest.Method = "POST";

            //Set Content-Type HTTP header to protobuf.
            firstRequest.ContentType = "application/x-google-protobuf";

            //Use MATLABParams class to make POST request body.
            MATLABParams mlParams = MATLABParams.Create(1, new List<Type> { typeof(Student[]) },
            mlParams.WriteTo(firstRequest.GetRequestStream());
            var response = (HttpWebResponse)firstRequest.GetResponse();

            Console.WriteLine("The HTTP status code of the request is " + response.StatusCode + "

            //If there is a MATLAB error, call to Result<>() throws MATLABException which contai
            Console.WriteLine("Parsing the result from the response of the HTTP Request using MA
            MATLABResult mlResult;
            mlResult = MATLABResult.Create(mlParams, response.GetResponseStream());
            try
            {
                Student[] result = mlResult.Result<Student[]>();
                Console.WriteLine("Printing the sorted Student array...\n");
                PrintStudent(result);
            }
            catch (MATLABException e)
            {
                Console.WriteLine(e.ToString());
            }
            Console.ReadKey();


        }

        static void PrintStudent(Student[] students)
        {
            foreach (Student s in students)
            {
                Console.WriteLine(s.ToString());
            }

        }
    }
}
```

See Also

POST Synchronous Request

More About

- “Marshal MATLAB Structures (structs) in C#” on page 2-21
- “Create a .NET MATLAB Production Server Client” on page 1-2
- “Create a C# Client” on page 1-3
- “Synchronous Execution”

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.