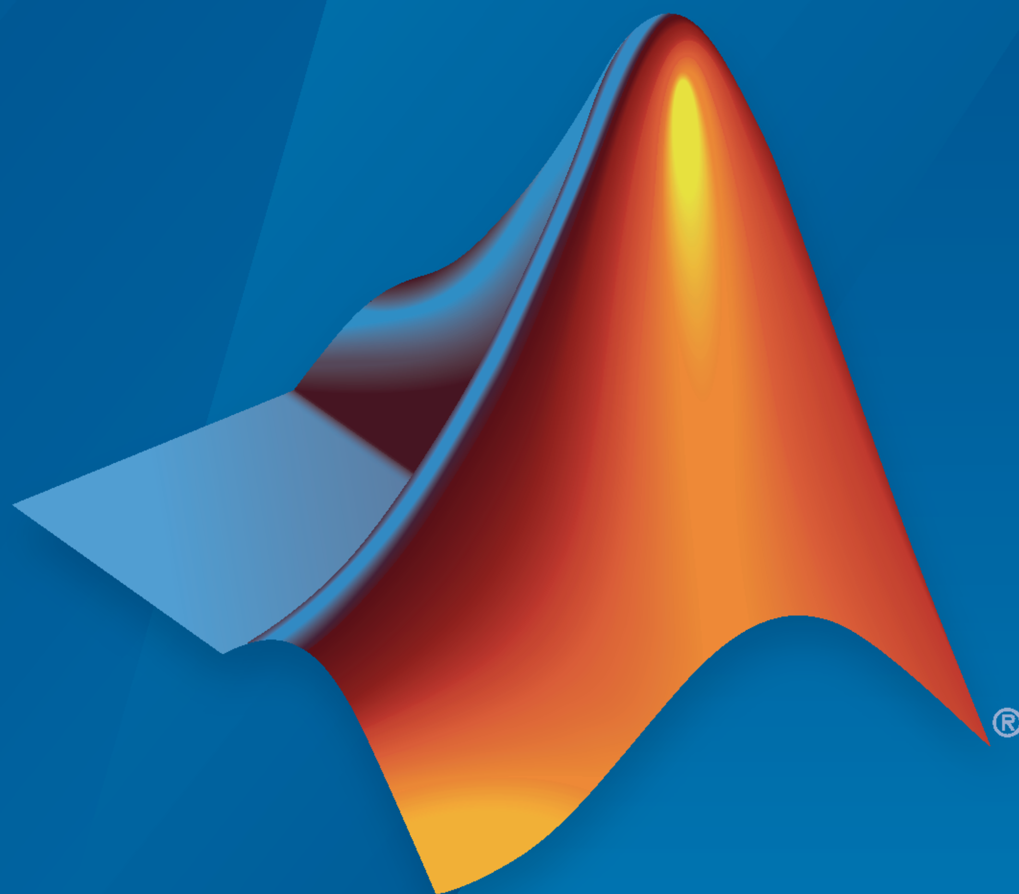
MATLAB® Production Server™

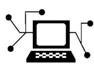
Java® Programming Guide



MATLAB®

R2021b

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™ Java® Programming Guide

© COPYRIGHT 2012–2021 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

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)

Contents

1 Client Programming

Create MATLAB Production Server Java Client Using MWHttpClient Class 1-2

Create Java Client Using MWHttpClient Class 1-3

Unsupported MATLAB Data Types for Client and Server Marshaling 1-6
- Supported Data Types 1-6
- Unsupported Data Types 1-6

2 Java Client Programming

Java Client Coding Best Practices 2-2
- Static Proxy Interface Guidelines 2-2
- Java Client Prerequisites 2-2
- Manage Client Lifecycle 2-2
- Handling Java Client Exceptions 2-3
- Managing System Resources 2-3
- Where to Find the Javadoc 2-4

Configure Client-Server Connection 2-5
- Default Configuration 2-5
- Implement Custom Connection Configurations 2-5

Invoke MATLAB Functions Dynamically 2-8
- Create a Proxy for Dynamic Invocation 2-8
- Invoke a MATLAB Function Dynamically 2-8
- Marshal MATLAB Structures 2-10

Bond Pricing Tool for Java Client 2-12
- Objectives 2-12
- Step 1: Write MATLAB Code 2-12
- Step 2: Create a Deployable Archive with the Production Server Compiler App 2-12
- Step 3: Share the Deployable Archive on a Server 2-13
- Step 4: Create the Java Client Code 2-13
- Step 5: Build the Client Code and Run the Example 2-15

Code Multiple Outputs for Java Client 2-16

Code Variable-Length Inputs and Outputs for Java Client 2-17

Marshal MATLAB Structures (Structs) in Java **2-18**
Marshaling a Struct Between Client and Server **2-18**

Data Conversion with Java and MATLAB Types **2-24**
Working with MATLAB Data Types **2-24**
Scalar Numeric Type Coercion **2-25**
Dimensionality in Java and MATLAB Data Types **2-25**
Empty (Zero) Dimensions **2-27**
Boxed Types **2-28**
Signed and Unsigned Types in Java and MATLAB Data Types **2-28**

Java Client Logging **2-29**
Use the Embedded log4j Engine **2-29**
Use an Existing Logging Engine **2-30**

Asynchronous RESTful Requests Using Protocol Buffers in the Java Client **2-31**
Deploy your MATLAB Function on the Server **2-31**
Make an Asynchronous Request to the Server **2-32**
Get the State Information of the Request **2-32**
View the Collection of Requests Owned by a Particular Client **2-33**
Retrieve the Results of a Request **2-33**

Synchronous RESTful Requests Using Protocol Buffers in the Java Client **2-37**
Deploy your MATLAB Function on the Server **2-37**
Make a Synchronous Request to the Server **2-38**
Receive and Interpret the Server Response **2-38**

Struct Support for RESTful Requests Using Protocol Buffers in the Java Client **2-41**
Deploy your MATLAB function on the server **2-41**
Create helper classes **2-42**
Make a synchronous request to the server **2-42**
Receive and interpret the server response **2-43**

Evaluate Deployed Machine Learning Models Using Java Client **2-46**
Determine Type of Input Argument for Deployed Function **2-46**
Represent Input Data as Array of Objects **2-46**
Write Client Application **2-51**

3 Security

Execute MATLAB Functions Using HTTPS **3-2**
Configure Client Environment for SSL **3-2**
Establish Secure Proxy Connection **3-3**
Establish Secure Connection Using Client Authentication **3-3**
Handle Exceptions **3-4**

Customize Security Configuration **3-8**
Specify Enabled Encryption Protocols **3-8**

Override Default Hostname Verification 3-9
Use Additional Server Authentication 3-10

A Data Conversion Rules

Conversion of Java Types to MATLAB Types A-2

Conversion of MATLAB Types to Java Types A-3

1

Client Programming

- “Create MATLAB Production Server Java Client Using MWHttpClient Class” on page 1-2
- “Create Java Client Using MWHttpClient Class” on page 1-3
- “Unsupported MATLAB Data Types for Client and Server Marshaling” on page 1-6

Create MATLAB Production Server Java Client Using MWHttpClient Class

To create a MATLAB Production Server client in Java:

1. Obtain `mps_client.jar` from *`$MPS_INSTALL`*`/client`. The client APIs are also available at MATLAB Production Server Client Libraries.
2. Configure your development environment to use `mps_client.jar`.
3. Based on your requirements, decide if the client uses a static proxy or a dynamic proxy.

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

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

     See "Invoke MATLAB Functions Dynamically" on page 2-8.
4. Write Java code to instantiate a proxy to a MATLAB Production Server instance and call the MATLAB functions.

   a. Create an `MWClient` object for communicating with the service hosted by a MATLAB Production Server instance.
   b. Create MATLAB data structures to hold the data passed between the client and server.
   c. Invoke MATLAB functions.
   d. Free system resources using the `close` method of the `MWClient` object.

See Also

More About

- "Create Java Client Using MWHttpClient Class" on page 1-3

Create Java Client Using MWHttpClient Class

This example shows how to write a MATLAB Production Server client using the Java client API. In your Java code, you will:

- Define a Java interface that represents the deployed MATLAB function.
- Instantiate a proxy object to communicate with the server.
- Call the deployed function in your Java code.

To create a Java MATLAB Production Server client application:

1. Create a new file, for example, `MPSClientExample.java`.
2. Using a text editor, open `MPSClientExample.java`.
3. Add the following import statements to the file:

```
import java.net.URL;
import java.io.IOException;
import com.mathworks.mps.client.MWClient;
import com.mathworks.mps.client.MWHttpClient;
import com.mathworks.mps.client.MATLABException;
```

4. Add a Java interface that represents the deployed MATLAB function.

   For example, consider the following `addmatrix` function deployed to the server:

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

a = a1 + a2;
```

   The interface for the `addmatrix` function follows.

```
interface MATLABAddMatrix {
       double[][] addmatrix(double[][] a1, double[][] a2)
         throws MATLABException, IOException;
   }
```

   When creating the interface, note the following:

   - You can give the interface any valid Java name.
   - You must give the method defined by this interface the same name as the deployed MATLAB function.
   - The Java method must support the same inputs and outputs supported by the MATLAB function, in both type and number. For more information about data type conversions and how to handle more complex MATLAB function signatures, see “Java Client Programming”.
   - The Java method must handle MATLAB exceptions and I/O exceptions.
5. Add the following class definition:

```
public class MPSClientExample
{
}
```

   This class now has a single `main` method that calls the generated class.
6. Add the `main()` method to the application.

```
public static void main(String[] args)
{
}
```

7 Add the following code to the top of the `main()` method to initialize the variables used by the application:

```
double[][] a1={{1,2,3},{3,2,1}};
double[][] a2={{4,5,6},{6,5,4}};
```

8 Instantiate a client object using the `MWHttpClient` constructor.

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

This class establishes an HTTP connection between the application and the server instance.

9 Call the `createProxy` method of the client object to create a dynamic proxy.

You must specify the URL of the deployable archive and the name of your interface `class` as arguments:

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

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

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

For more information about the `createProxy` method, see the Javadoc included in the *`$MPS_INSTALL`*`/client` folder, where *`$MPS_INSTALL`* is the name of your MATLAB Production Server installation folder.

10 Call the deployed MATLAB function in your Java application by calling the public method of the interface.

```
  double[][] result = m.addmatrix(a1,a2);
```

11 Call the `close()` method of the client object to free system resources.

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

12 Save the Java file.

The completed Java file should resemble the following:

```
import java.net.URL;
import java.io.IOException;
import com.mathworks.mps.client.MWClient;
import com.mathworks.mps.client.MWHttpClient;
import com.mathworks.mps.client.MATLABException;

interface MATLABAddMatrix
  {
    double[][] addmatrix(double[][] a1, double[][] a2)
         throws MATLABException, IOException;
  }

public class MPSClientExample {

    public static void main(String[] args){

        double[][] a1={{1,2,3},{3,2,1}};
        double[][] a2={{4,5,6},{6,5,4}};

        MWClient client = new MWHttpClient();
```

```
        try{
            MATLABAddMatrix m = client.createProxy(new URL("http://localhost:9910/addmatrix"),
                                                   MATLABAddMatrix.class);
            double[][] result = m.addmatrix(a1,a2);

            // Print the resulting matrix
            printResult(result);

        }catch(MATLABException ex){

            // This exception represents errors in MATLAB
               System.out.println(ex);
        }catch(IOException ex){

            // This exception represents network issues.
               System.out.println(ex);
        }finally{

            client.close();
        }
    }

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

13 Compile the Java application, using the `javac` command or use the build capability of your Java IDE.

For example, enter the following at the system command prompt:

```
javac -classpath "MPS_INSTALL_ROOT\client\java\mps_client.jar" MPSClientExample.java
```

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

For example, enter the following at the system command prompt:

```
java -classpath .;"MPS_INSTALL_ROOT\client\java\mps_client.jar" MPSClientExample
```

The application returns the following at the console:

```
5.0 7.0 9.0
9.0 7.0 5.0
```

See Also

More About

- “Bond Pricing Tool for Java Client” on page 2-12
- “Create MATLAB Production Server Java Client Using MWHttpClient Class” on page 1-2

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”

2

Java Client Programming

- “Java Client Coding Best Practices” on page 2-2
- “Configure Client-Server Connection” on page 2-5
- “Invoke MATLAB Functions Dynamically” on page 2-8
- “Bond Pricing Tool for Java Client” on page 2-12
- “Code Multiple Outputs for Java Client” on page 2-16
- “Code Variable-Length Inputs and Outputs for Java Client” on page 2-17
- “Marshal MATLAB Structures (Structs) in Java” on page 2-18
- “Data Conversion with Java and MATLAB Types” on page 2-24
- “Java Client Logging” on page 2-29
- “Asynchronous RESTful Requests Using Protocol Buffers in the Java Client” on page 2-31
- “Synchronous RESTful Requests Using Protocol Buffers in the Java Client” on page 2-37
- “Struct Support for RESTful Requests Using Protocol Buffers in the Java Client” on page 2-41
- “Evaluate Deployed Machine Learning Models Using Java Client” on page 2-46

Java Client Coding Best Practices

Static Proxy Interface Guidelines

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

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

Java Client Prerequisites

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

1. Install a Java IDE of your choice. Follow instructions on the Oracle Web site for downloading Java , if needed.
2. Add `mps_client.jar` (located in *$MPS_INSTALL*`\client\java`) to your Java **`CLASSPATH`** and Build Path. This JAR file is sometimes defined in separate GUIs, depending on your IDE.

   Generate one deployable archive into your server's `auto_deploy` folder for each MATLAB application you plan to deploy. For information about creating a deployable archive with the Production Server Compiler app, see "Create Deployable Archive for MATLAB Production Server".

   Your server's `main_config` file should point to where your MATLAB Runtime instance is installed.
3. The server hosting your deployable archive must be running.

Manage Client Lifecycle

A single Java client connects to one or more servers available at various URLs. Even though you create multiple instances of `MWHttpClient` on page 1-2, one instance is capable of establishing connections with multiple servers.

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

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

Locally Scoped Instance

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

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

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

Servlet Implementation

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

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

  protected void doGet(HttpServletRequest req,HttpServletResponse resp)
  throws ServletException,java.io.IOException
  {
    // Code that uses client to communicate with the server
  }

  public void destroy()
  {
    client.close();
  }
}
```

Handling Java Client Exceptions

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

Java Client Exceptions

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

Managing System Resources

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

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

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

Where to Find the Javadoc

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

Configure Client-Server Connection

The MWHttpClientConfig interface in the Java client API defines the default configuration that an instance of MWHttpClient uses when it establishes a client-server connection. To modify the default configuration, extend the MWHttpClientDefaultConfig class and override its methods.

Default Configuration

The default configuration consists of the following fields. The MWHttpClientDefaultConfig class inherits these fields from the MWHttpClientConfig interface.

Field Name	Description	Default Value
DEFAULT_IS_COOKIE_ENABLED	Determines if the client sets the HTTP cookie.	true
DEFAULT_IS_INTERRUPTABLE	Determines if the client can interrupt MATLAB function execution.	false
DEFAULT_RESPONSE_SIZE_LIMIT	Maximum size, in bytes, of the response that a client accepts.	64*1024*1024 (64 MB)
DEFAULT_NUM_CONNECTIONS_PER_ADDRESS	Maximum number of connections that the client opens to fulfill multiple requests.	-1, specifies that the client can uses as many connections as the system allows.
DEFAULT_TIMEOUT_MS	Amount of time, in milliseconds, that the client waits for a server response before timing out.	120000

Implement Custom Connection Configurations

To implement a custom client-server connection configuration, extend the MWHttpClientDefaultConfig class, and override its methods to provide an implementation for your custom configuration. The MWHttpClientDefaultConfig class has one getter method corresponding to each configuration field that you can override.

Method	Description
public boolean isCookieEnabled()	Returns true if the client sets the HTTP cookie; otherwise, returns false.
public boolean isInterruptible()	Returns true if the client can interrupt the execution of a deployed MATLAB function while waiting for a response; otherwise, returns false.
public int getResponseSizeLimit()	Returns the maximum number of bytes that the client can accept in a server response.
public int getTimeoutMs()	Returns the time in milliseconds that a client waits for a response before generating an error.

Method	Description
public in getMaxConnectionsPerAddress()	Returns the maximum number of connections that a client can use to handle simultaneous requests.

Note If `isInterruptible()` returns `false`, then `getMaxConnectionsPerAddress()` must return `-1`.

To change one or more client-server connection properties:

1. Implement a custom connection configuration by extending the `MWHttpClientDefaultConfig` class.
2. Create the client-server connection using the `MWHttpClient` constructor that accepts an instance of `MWHttpClientDefaultConfig`.

You need to override only the getters for the properties that you want to change. For example, to specify that a client times out after six seconds, can accept 4 MB responses, and does not save HTTP cookies, override `getTimeOutMs()`, `getResponseSizeLimit()`, and `isCookieEnabled()`. The sample code follows.

```
//Implement custom configuration
class MyClientConfig extends MWHttpClientDefaultConfig
{
  public long getTimeOutMs()
  {
    return 6000;
  }
  public int getResponseSizeLimit()
  {
    return 4*1024*1024;
  }
  public boolean isCookieEnabled()
  {
    return false;
  }
}
...
//Create client-server connection
MWClient client = new MWHttpClient(new MyClientConfig());
...
```

To modify the security configuration, provide an object that extends the `MWSSLDefaultConfig` utility class as an argument to the `MWHttpClient` constructor.

```
...
MWHttpClient(MWHttpClientConfig config, MWSSLConfig sslConfig)
...
```

For more information, see “Customize Security Configuration” on page 3-8.

See Also

More About

- “Create MATLAB Production Server Java Client Using MWHttpClient Class” on page 1-2
- “Create Java Client Using MWHttpClient Class”
- “Customize Security Configuration” on page 3-8

Invoke MATLAB Functions Dynamically

In this section...

"Create a Proxy for Dynamic Invocation" on page 2-8
"Invoke a MATLAB Function Dynamically" on page 2-8
"Marshal MATLAB Structures" on page 2-10

To dynamically invoke functions on an MATLAB Production Server instance, you use a reflection-based proxy to construct the MATLAB function request. The function name and all of the inputs and outputs are passed as parameters to the method invoking the request. This means that you do not need to recompile your application every time you add a function to a deployed archive.

To dynamically invoke a MATLAB function:

1. Instantiate an instance of the `MWHttpClient` class.
2. Create a reflection-based proxy object using one of the `createComponentProxy()` methods of the client connection.
3. Invoke the function using one of the `invoke()` methods of the reflection-based proxy.

Create a Proxy for Dynamic Invocation

A reflection-based proxy implements the `MWInvokable` interface and provides methods that enables you to directly invoke any MATLAB function in a deployable archive. As with the interface-based proxy, the reflection-based proxy is created from the client connection object. The `MWHttpClient` class 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, MWMarshalingRules marshalingRules)` creates a proxy that uses structures.

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

```
MWClient myClient = new MWHttpClient();

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

Invoke a MATLAB Function Dynamically

A reflection-based proxy has three methods for invoking functions on a server:

- `Object[] invoke(final String functionName, final int nargout, final Class<T> targetType, final Object... inputs)` invokes a function that returns `nargout` values.
- `<T> T invoke(final String functionName, final Class<T> targetType, final Object... inputs)` invokes a functions that returns a single value.
- `invokeVoid(final String functionName, final Object... inputs)` invokes a function that returns no values.

All methods map to the MATLAB function as follows:

- First argument is the function name
- Middle set of arguments, `nargout` and `targetType`, represent the return values of the function
- 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 Java client, use the `invoke()` method that takes the number of return arguments:

```
double[] myRange = new double[]{2,5,7,100,0.5};
try
{
  Object[] myLimits = myProxy.invoke("myLimits",
                                     2,
                                     Object[].class,
                                     myRange);
  double myMin = ((Double) myLimits[0]).doubleValue();
  double myMax = ((Double) myLimits[1]).doubleValue();
  System.out.printf("min: %f max: %f",myMin,myMax);
}
catch (Throwable e)
{
  e.printStackTrace();
}
```

Because Java cannot determine the proper types for each of the returned values, this form of `invoke` always returns `Object[]` and always takes `Object[].class` as the target type. You must cast the returned values into the proper types.

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 Java client, use the `invoke()` method that does not take the number of return arguments:

```
double[][] a1={{1,2,3},{3,2,1}};
double[][] a2={{4,5,6},{6,5,4}};
try
{
  Double[][] result = myProxy.invoke("addmatrix",
                                     Double[][].class,
                                     a1,
                                     a2);
```

```
  for(Double[] row : result)
  {
    for(double element : row)
    {
      System.out.print(element + " ");
    }
  }
} catch (Throwable e)
{
  e.printStackTrace();
}
```

Return No Outputs

The MATLAB function `foo` does not return value.

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

To invoke `foo` from a Java client, use the `invokeVoid()` method:

```
double[][] a={{1,2,3},{3,2,1}};
try
{
  myProxy.invokeVoid("foo", (Object)a);
}
catch (Throwable e)
{
  e.printStackTrace();
}
```

Marshal MATLAB Structures

If any MATLAB function in a deployable archive uses structures, you need to provide marshaling rules to the reflection-based proxy. To provide marshaling rules to the proxy:

1. Implement a new set of marshaling rules by extending the `MWDefaultMarshalingRules` interface to use a list of the classes being marshaled.
2. Create the proxy using the `createComponentProxy(URL archiveURL, MWMarshalingRules marshalingRules)` method.

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

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

Java 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 `studentMarshaler`.

```
class studentMarshaler extends MWDefaultMarshalingRules
{
  public List<Class> getStructTypes() {
    List structType = new ArrayList<Class>();
```

```
    structType.add(Student.class);
    return structType;
  }
}
```

Create the proxy for `studentChecker` by passing `studentMarshaler` 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 Java" on page 2-18.

Bond Pricing Tool for Java Client

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

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

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

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

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

Objectives

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

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

Step 1: Write MATLAB Code

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

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

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

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

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

end
```

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

To create the deployable archive for this example:

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

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

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

Step 3: Share the Deployable Archive on a Server

1. Download the MATLAB Runtime, if needed, at https://www.mathworks.com/products/compiler/mcr. See "Supported MATLAB Runtime Versions" for more information.
2. Create a server using `mps-new`. See "Create Server Instance" for more information.
3. If you have not already done so, specify the location of the MATLAB Runtime to the server by editing the server configuration file, `main_config` and specifying a path for `--mcr-root`. See "Configure Server" for details.
4. "Start Server Instance" and "Verify Server Status".
5. Copy the `BondTools.ctf` file to the `auto_deploy` folder on the server for hosting.

Step 4: Create the Java Client Code

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

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

This Java code...	Provides this functionality...
`BondPricingTool.java`	Runs the calculator application. The variable values of the pricing function are declared in this class.
`BondTools.java`	Defines `pricecalc` method interface, which is later used to connect to a server to invoke `pricecalc.m`
`BondToolsFactory.java`	Factory that creates new instances of `BondTools`
`BondToolsStub.java`	Java class that implements a dummy `pricecalc` Java method. Creating a stub method is a technique that allows for calculations and processing to be added to the application at a later time.
`BondToolsStubFactory.java`	Factory that returns new instances of `BondToolsStub`
`RequestSpeedMeter.java`	Displays a GUI interface and accepts inputs using Java Swing classes
`ServerBondToolsFactory.java`	Factory that creates new instances of `MWHttpClient` and creates a proxy that provides an implementation of the `BondTools` interface and allows access to `pricecalc.m`, hosted by the server

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

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

Declare Java Method Signatures Compatible with MATLAB Functions You Deploy

To use the MATLAB functions you defined in “Step 1: Write MATLAB Code” on page 2-12, declare the corresponding Java method signature in the interface BondTools.java:

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

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

Instantiate MWClient, Create Proxy, and Specify Deployable Archive

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

1. Instantiate MWClient with an instance of MWHttpClient:

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

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

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

Use dispose() Consistently to Free System Resources

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

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

For example, note that in ServerBondToolsFactory.java, you dispose of the MWHttpClient instance you created in “Instantiate MWClient, Create Proxy, and Specify Deployable Archive” on page 2-14 when it is no longer needed.

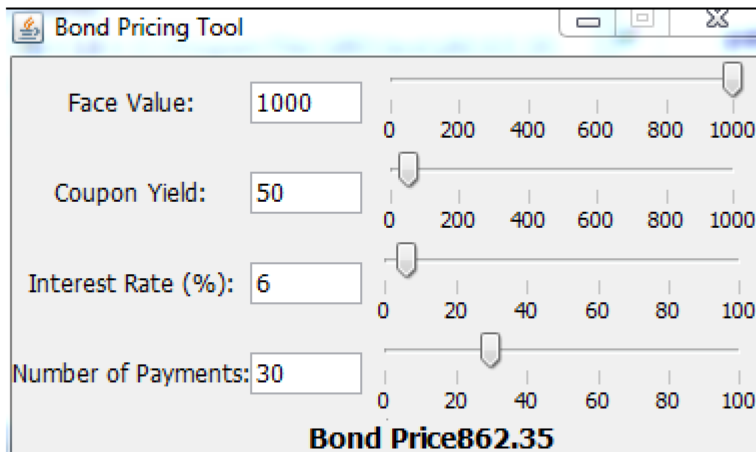
Additionally, note the dispose() calls to clean up the factories in BondToolsStubFactory.java and BondTools.java.

Step 5: Build the Client Code and Run the Example

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

- Added `mps_client.jar` (*$MPS_INSTALL*\client\java) to your Java **CLASSPATH** and Build Path.
- Copied your deployable archive to your server's `auto_deploy` folder.
- Modified your server's `main_config` file to point to where your MATLAB Runtime is installed.
- "Start Server Instance" and "Verify Server Status".

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



Code Multiple Outputs for Java Client

MATLAB allows users to write functions that return multiple outputs.

For example, consider this MATLAB function signature:

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

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

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

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

In this case, the matching signature in Java is:

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

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

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

Note the following coding best practices illustrated by this example:

- Both the MATLAB function signature and the Java method signature using the name `multipleOutputs`. Both signatures define two inputs and two outputs.
- MATLAB Java interface supports direct conversion from Java double array to MATLAB double array and from Java string to MATLAB char array. For more information, see “Conversion of Java Types to MATLAB Types” on page A-2 and “Conversion of MATLAB Types to Java Types” on page A-3.

For more information, see “Java Client Coding Best Practices” on page 2-2.

Code Variable-Length Inputs and Outputs for Java Client

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

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

For example, consider this MATLAB function:

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

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

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

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

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

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

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

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

Note the following coding best practices illustrated by this example:

- Both the MATLAB function signature and the Java method signature using the name `vararginout`. Both signatures define two inputs and two outputs.
- MATLAB Java interface supports direct conversion from Java double array to MATLAB double array and from Java string to MATLAB char array. For more information, see “Conversion of Java Types to MATLAB Types” on page A-2 and “Conversion of MATLAB Types to Java Types” on page A-3.

Marshal MATLAB Structures (Structs) in Java

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

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

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

In MATLAB, a structure is created as follows:

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

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

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

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

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

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

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

Marshaling a Struct Between Client and Server

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

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

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

In this example, MATLAB function `sortstudents` takes in an array of structures (see “Marshal MATLAB Structures (Structs) in Java” on page 2-18 for details).

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

```
function sorted = sortstudents(unsorted)
% Receive a vector of students as input
% Get scores of all the students
scores = {unsorted.score};
```

```
% Convert the cell array containing scores into a numeric array or doubles
scores = cell2mat(scores);
% Sort the scores array
[s i] = sort(scores);
% Sort the students array based on the sorted scores array
sorted = unsorted(i);
```

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

You package `sortstudents` into a deployable archive (`scoresorter.ctf`) using the Production Server Compiler app (see "Create Deployable Archive for MATLAB Production Server" for details) and make it available on the server at `http://localhost:9910/scoresorter` for access by the Java client (see "Share Deployable Archive").

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

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

Java Class Student

```
public class Student{

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

    public Student(){
    }

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

    public String getName(){
        return name;
    }

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

    public int getScore(){
        return score;
    }

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

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

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

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

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

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

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

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

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

1. Create `MWHttpClient` and associated proxy (using `createProxy`) as shown in “Create Java Client Using MWHttpClient Class” on page 1-3.
2. Create an unsorted student struct array in Java that mimics the MATLAB struct in naming, number of inputs and outputs, and type validity in MATLAB. See “Java Client Coding Best Practices” on page 2-2 for more information.
3. Sort the student array and display it.

```
import java.net.URL;
import java.io.IOException;
import com.mathworks.mps.client.MWClient;
import com.mathworks.mps.client.MWHttpClient;
import com.mathworks.mps.client.MATLABException;
import com.mathworks.mps.client.annotations.MWStructureList;

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

public class ClientExample {

    public static void main(String[] args){

        MWClient client = new MWHttpClient();
        try{
            StudentSorter s =
               client.createProxy(new URL("http://localhost:9910/scoresorter"),
                                                     StudentSorter.class );
            Student[] students = new Student[]{new Student("Toni Miller", 90, "A"),
                                               new Student("Ed Plum",     80, "B+"),
                                               new Student("Mark Jones",  85, "A-")};
```

```
            Student[] sorted = s.sortstudents(students);
            System.out.println("Student list sorted in the 
                            ascending order of scores : ");
            for(Student st:sorted){
                System.out.println(st);
            }
        }catch(IOException ex){
            System.out.println(ex);
        }catch(MATLABException ex){
            System.out.println(ex);
        }finally{
            client.close();
        }
    }
}
```

Map Java Field Names to MATLAB Field Names

Java classes that represent MATLAB structures use the Java Beans `Introspector` class (https://docs.oracle.com/javase/6/docs/api/java/beans/Introspector.html) to map properties to fields and its default naming conventions are used.

This means that by default its `decapitalize()` method is used. This maps the first letter of a Java field into a lower case letter. By default, it is not possible to define a Java field which will map to a MATLAB field which starts with an upper case.

You can override this behavior by implementing a `BeanInfo` class with a custom `getPropertyDescriptors()` method. For example:

```
import java.beans.IntrospectionException;
import java.beans.PropertyDescriptor;
import java.beans.SimpleBeanInfo;
public class StudentBeanInfo extends SimpleBeanInfo
{
  @Override
  public PropertyDescriptor[] getPropertyDescriptors()
  {
  PropertyDescriptor[] props = new  PropertyDescriptor[3];
  try
  {
    // name uses default naming conventions so we do not need to
    // explicitly specify the accessor names.
    props[0] = new PropertyDescriptor("name",MyStruct.class);
    // score uses default naming conventions so we do not need to
    // explicitly specify the accessor names.
    props[1] = new PropertyDescriptor("score",MyStruct.class);
    // Grade uses a custom naming convention so we do need to
    // explicitly specify the accessor names.
    props[1] = new PropertyDescriptor("Grade",MyStruct.class,
                                      "getGrade","setGrade");
    return props;
  }
  catch (IntrospectionException e)
  {
    e.printStackTrace();
  }

  return null;
  }
}
```

Defining MATLAB Structures Only Used as Inputs

When defining Java structs as inputs, follow these guidelines:

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

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

```
public class Student{

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

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

    public String getName(){
        return name;
    }

    public int getScore(){
        return score;
    }

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

Defining MATLAB Structures Only Used as an Output

When defining Java structs as outputs, follow these guidelines:

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

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

```
public class Student{

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

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

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

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

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

```
public class Student{

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

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

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

Defining MATLAB Structures Used as Both Inputs and Outputs

If the `Student` class is used as both an input and output, you need to provide `get` methods to perform marshaling to MATLAB. For marshaling from MATLAB, use `set` methods or `@ConstructorProperties` annotation on page 2-22.

Data Conversion with Java and MATLAB Types

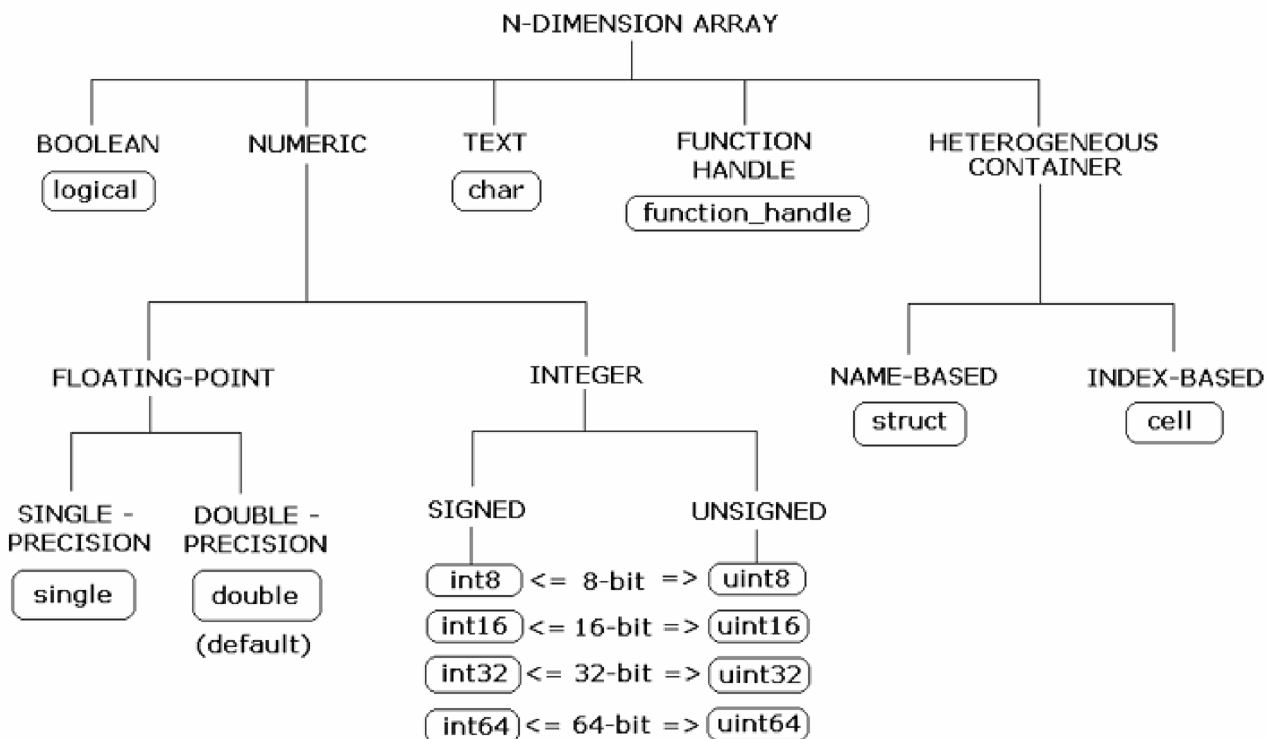
Working with MATLAB Data Types

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

All of the fundamental MATLAB classes are circled in the diagram “Fundamental MATLAB Data Types” on page 2-24.

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

Note Function Handles are not supported by MATLAB Production Server.



Fundamental MATLAB Data Types

The expected conversion results for Java to MATLAB types are listed in “Conversion of Java Types to MATLAB Types” on page A-2. The expected conversion results for MATLAB to Java types are listed in “Conversion of MATLAB Types to Java Types” on page A-3.

Scalar Numeric Type Coercion

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

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

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

MATLAB to Java Numeric Type Compatibility

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

Dimensionality in Java and MATLAB Data Types

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

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

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

Dimension Coercion

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

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

Padding

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

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

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

How MATLAB Pads Your Java Method Return Type

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

Padding Dimensions in MATLAB and Java Data Conversion

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

Truncation

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

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

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

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

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

How MATLAB Truncates Your Java Method Return Type

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

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

Truncating Dimensions in MATLAB and Java Data Conversion

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

Empty (Zero) Dimensions

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

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

Passing Java Empties to MATLAB

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

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

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

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

Passing MATLAB Empties to Java

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

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

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

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

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

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

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

Boxed Types

Boxed Types are used to wrap opaque C structures.

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

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

For example, the following method signatures work interchangeably:

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

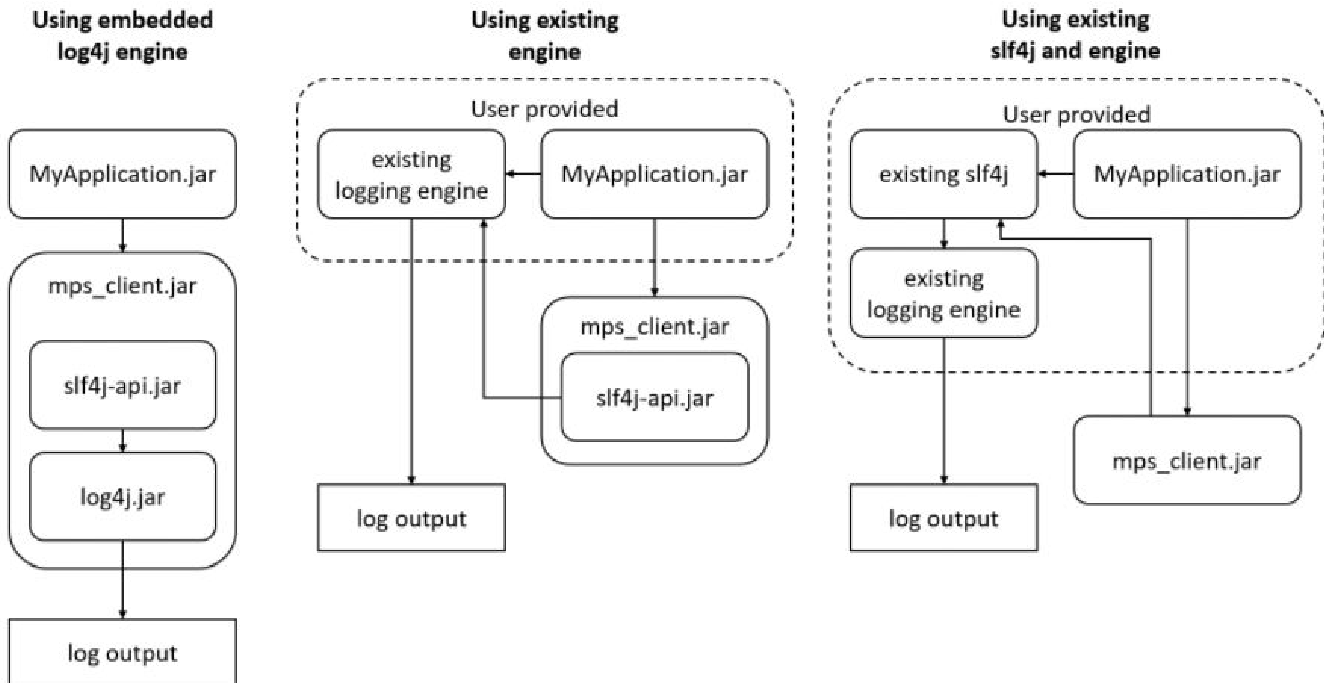
Signed and Unsigned Types in Java and MATLAB Data Types

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

Java Client Logging

Logging capability is available in the Java client to record details such as HTTP request statuses, server URLs, and output data. Logging is implemented using the `slf4j`, so it can work with multiple logging engines, such as `log4j`, `logback`, or `java.util.logging`.

It can utilize the logging engine used in your project, from one of the `slf4j` supported engines, or load its own embedded engine if none is provided.



Use the Embedded log4j Engine

When your project does not use a logging engine, and you want to log just the Java client activity, you can activate the Java client embedded `log4j` engine it can use once activated. To use the embedded engine, pass in a `log4j` configuration file to the Java application at startup. To do this, add the file location URL to the `log4j.configuration` JVM property. The URL to a file on the file system is:

```
file:/path/to/file/filename
```

The embedded engine is loaded only if no engine is provided.

The default `log4j` configuration file that outputs to standard out is found at the following location: `$MPS_INSTALL/client/java/log4j.properties`.

Example (UNIX® syntax):

```
java -cp ./mps_client.jar:./Magic.jar -Dlog4j.configuration=file:/$MPS_INSTALL/client/java/log4
```

Use an Existing Logging Engine

If your project uses an existing engine, the Java client can use that engine for logging. Your project can use any engine that supports `slf4j`. To use an existing engine, you must be able to load it into your Java application, and it must be on your Java classpath. If you need different version of the `slf4j` engine, you can load your own `slf4j` library and include it in your classpath.

For `java.util.logging`, you need to load and use the `java.util.logging.Logger` class in your Java application code before the `com.mathworks.mps.client.MWHttpClient` class is loaded.

For logback, add both the `logback-classic` and `logback-core` jar files onto the classpath.

If you encounter version mismatch issues between your engine and `slf4j`, it is best to load your own `slf4j-api.jar` of the appropriate version by setting it on the Java classpath. This situation can occur if you are using later versions of `logback`.

Example (UNIX syntax):

```
#Using existing log4j engine
java -cp ./log4j.jar:./mps_client.jar:./MyApplication.jar -Dlog4j.configuration=file:/path/to/log

#Using existing logback engine
java -cp ./logback-classic.jar:./logback-core.jar:./mps_client.jar:./MyApplication.jar -Dlogback

#Using existing slf4j API
java -cp ./slf4j-api.jar:./mps_client.jar:./MyApplication.jar MainClass

#Using existing logback engine with existing slf4j
java -cp ./slf4j-api.jar:./logback-classic.jar:./logback-core.jar:./mps_client.jar:./MyApplicatio
```

Refer to the third-party logging engine documentation for more information on how to configure the logging behavior.

Note If loading existing `slf4j` or `logback` jars, it must be set in front of the `mps_client.jar` on the Java classpath.

See Also

Asynchronous RESTful Requests Using Protocol Buffers in the Java Client

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

To use protobuf when making a request to the server, set the HTTP `Content-Type` header to `application/x-google-protobuf` in the client code. The Java 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 Java client library provides methods and classes to deserialize the protobuf responses.

To use the Java client library, you must include `mps_client.jar` in the `CLASSPATH`.

The following table shows where to find the `mps_client.jar` file, Javadoc, and sample code for the example.

Location of `mps_client.jar`	• *MPS_INSTALL*/client/java • *MATLABProductionServer_<release>_Clients*/java
Location of Javadoc	• *MPS_INSTALL*/client/java/doc • *MATLABProductionServer_<release>_Clients*/java/doc
Location of code for the example files	• *MPS_INSTALL*/client/java/examples • *MATLABProductionServer_<release>_Clients*/java/examples/MagicSquare
• *MPS_INSTALL* is the location in which MATLAB Production Server is installed. • *MATLABProductionServer_<release>_Clients* is the folder containing MATLAB Production Server client libraries that you can download from https://www.mathworks.com/products/matlab-production-server/client-libraries.html.	

The example uses the `java.net` package for making HTTP requests to evaluate a MATLAB function deployed on a MATLAB Production Server instance running on `http://localhost:9910`.

Deploy your MATLAB Function on the Server

Write a MATLAB function `mymagic` that uses the `magic` function to create a magic square, then deploy it on the server.

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

```
function m = mymagic(in)

  m = magic(in);
end
```

The function `mymagic` takes a single `int32` input and returns a magic square as a 2-D `double` array.

Make an Asynchronous Request to the Server

1. Construct the request URL.

   In the Java client, use the POST Asynchronous Request to make the initial request to the server. The request URL comprises of the address of the server instance, the name of the deployed archive and the name of the MATLAB function to evaluate. Set the HTTP request `mode` to `async` and `client` to a user-defined identifier value in the query parameters.

   ```
   String clientId = "123";
   String mpsBaseUrl = "http://localhost:9910";
   URL url;
   url = new URL(mpsBaseUrl + "/mymagic/mymagic?mode=async&client="+clientId);
   ```

2. Set the request headers.

   Set the HTTP `Content-Type` header to `application/x-google-protobuf`, as the API returns a byte array of protocol buffer messages.

   ```
   final static protected String CONTENT_TYPE = "application/x-google-protobuf";
   HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
   urlConnection.setDoOutput(true);
   urlConnection.setRequestProperty("Content-Type", CONTENT_TYPE);
   ```

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

   Use the `newInstance(arg1, arg2, arg3)` method defined in the `MATLABParams` class to build the protocol buffer message. Since the `mymagic` function returns a single 2-D array, set `arg1` to `1` and `arg2` to `double[][].class`. Specify an integer value for `arg3`, which is the input to the `mymagic` function.

   ```
   MATLABParams mlMakeBody = MATLABParams.newInstance(1, double[][].class, 2);
   ```

4. Send the request to the server.

   Write the `MATLABParams` `mlMakeBody` object to the output stream of the HTTP request.

   ```
   OutputStream output = urlConnection.getOutputStream();
   output.write(mlMakeBody.getRequestBody());
   output.flush();
   ```

5. 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 mlInitialResponse =
       MATLABRequestHandle.newInstance(urlConnection.getInputStream());
   System.out.println("First Request has been Sent. Inital response is below");
   System.out.println("State: "+ mlInitialResponse.getState() + " " + "Request URL: "
       +mlInitialResponse.getRequestURL() + " Last modified sequence: " +
           mlInitialResponse.getLastModifiedSeq());
   ```

Get the State Information of the Request

1. Make a request to get the request state information.

Use the GET State Information RESTful API to get the state of the request. In the request URL, set the query parameter `format` to `protobuf`, so that the server returns the output in protocol buffer format.

```
url = new URL(mpsBaseUrl + mlInitialResponse.getRequestURL() + "/info?" + "format=protobu
urlConnection = (HttpURLConnection) url.openConnection();
urlConnection.setRequestProperty("Content-Type", CONTENT_TYPE);
urlConnection.setRequestMethod("GET");
urlConnection.connect();
```

2 Parse the response.

Parse the response using methods defined in the `MATLABRequest` class to get the state of the request and the current `lastModifiedSeq` value at the server.

```
MATLABRequest requestInfoTmp = MATLABRequest.newInstance(urlConnection.getInputStream());
System.out.println("State: "+requestInfoTmp.getState() + " Last modified sequence: " + re
```

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.

View the Collection of Requests Owned by a Particular Client

Use the GET Collection of Requests RESTful API to view information about multiple requests sent by a particular client represented by `clientId`. In the request URL, set the query parameter `format` to `protobuf`, so that the server returns the output in protocol buffer format. Use the `MATLABRequests` class `newInstance` method to parse the response body of a successful request. The `MATLABRequests` class has a `getMATLABRequests` method that returns a `Map` of `requestURL` and the `MATLABRequest` object.

```
url = new URL(mpsBaseUrl + mlInitialResponse.getInstanceId() + "requests" + "?since="
    + mlInitialResponse.getLastModifiedSeq() + "&format=protobuf&" + "clients=" + clientId);
urlConnection = (HttpURLConnection) url.openConnection();
urlConnection.setRequestProperty("Content-Type", CONTENT_TYPE);
urlConnection.setRequestMethod("GET");
urlConnection.connect();

MATLABRequests updates = MATLABRequests.newInstance(urlConnection.getInputStream());

Map<String, MATLABRequest> urlUpdates = updates.getMATLABRequests();
System.out.println("State of the Requests with the client: " + clientId);
for (String requestURL : urlUpdates.keySet()) {
    System.out.println(requestURL + ":" + urlUpdates.get(requestURL).getState());
}
```

Retrieve the Results of a 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.

```
url = new URL(mpsBaseUrl + mlInitialResponse.getRequestURL() + "/result?" + "format=proto
urlConnection = (HttpURLConnection) url.openConnection();
```

```
    urlConnection.setRequestProperty("Content-Type", CONTENT_TYPE);
    urlConnection.setRequestMethod("GET");
    urlConnection.connect();
```

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, pass the MATLABParams mlMakeBody object and the response body of the GET Result of Request request to the newInstance method.

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

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

```
    if (requestInfoTmp.compareTo(MATLABRequestState.ERROR_STATE) == 0) {
        HTTPErrorInfo httpErrorInfo = HTTPErrorInfo.newInstance(urlConnection.getInputStream(
        System.out.println("ErrorCode: " + httpErrorInfo.getHttpErrorCode());
        System.out.println("Error Message: " + httpErrorInfo.getHttpErrorMessage());
        System.out.println("Error body: " + httpErrorInfo.getHttpBody());
    }
    else{
        MATLABResult<double[][]> mlFinalResult1 = MATLABResult.newInstance(mlMakeBody, urlCor
        try{
            double[][] magicSq1 = mlFinalResult1.getResult();
            printResult(magicSq1);
        }catch(MATLABException e){
            e.printStackTrace();
        }
    }
```

3 Display the results.

   Write a helper method printResult that takes as input the result that is parsed from the response body and prints the corresponding 2-D array.

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

Sample code for the AsyncExample.java Java client follows.

Code:

AsyncExample.java

```
import com.mathworks.mps.client.MATLABException;
import com.mathworks.mps.client.rest.*;

import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;
```

```
public class AsyncExample{

    final static protected String CONTENT_TYPE = "application/x-google-protobuf";

    public static void main(String[] args){

        try{
            String clientId = "123";

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

         // Use the java.net package's URLConnection as HTTP Client in this example.
            URL url;
            url = new URL(mpsBaseUrl + "/mymagic/mymagic?mode=async&client="+clientId);

            HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
            urlConnection.setDoOutput(true);
            //Set Content-Type to protobuf.
            urlConnection.setRequestProperty("Content-Type", CONTENT_TYPE);

            // Make the initial POST request body with MATLABParams class.
            MATLABParams mlMakeBody = MATLABParams.newInstance(1, double[][].class, 2);

            // Write the MATLABParams object into the output stream of the HTTP Request.
            OutputStream output = urlConnection.getOutputStream();
            output.write(mlMakeBody.getRequestBody());
            output.flush();

            // Parse the response body of the above HTTP request with methods from the MATLABResu
            // to retrieve the request URL, lastModified value and state of the request.
            MATLABRequestHandle mlInitialResponse = MATLABRequestHandle.newInstance(urlConnectio
            System.out.println("First Request has been Sent. Initial response is below");
            System.out.println("State: "+ mlInitialResponse.getState() + " " + "Request URL: "+ml

            // Query for the state of the request.
            url = new URL(mpsBaseUrl + mlInitialResponse.getRequestURL() + "/info?" + "format=pro
            urlConnection = (HttpURLConnection) url.openConnection();
            urlConnection.setRequestProperty("Content-Type", CONTENT_TYPE);
            urlConnection.setRequestMethod("GET");
            urlConnection.connect();

            MATLABRequest requestInfoTmp = null;
            // Parse the response body using methods from MATLABRequest class.
            requestInfoTmp = MATLABRequest.newInstance(urlConnection.getInputStream());
            System.out.println("State: "+requestInfoTmp.getState() + " Last modified sequence: "

          // Loop to check if the state of the request is READY_STATE.
            for (int i = 0; i < 20; i++) {
                url = new URL(mpsBaseUrl + mlInitialResponse.getRequestURL() + "/info?" + "forma
                urlConnection = (HttpURLConnection) url.openConnection();
                urlConnection.setRequestProperty("Content-Type", CONTENT_TYPE);
                urlConnection.setRequestMethod("GET");
                urlConnection.connect();

                // Parse the response body using methods from MATLABRequest class.
```

```
                requestInfoTmp = MATLABRequest.newInstance(urlConnection.getInputStream());
                System.out.println("State: "+requestInfoTmp.getState() + " Last modified sequence
                Thread.sleep(1000);
            }

            // Once the state changes to READY_STATE, query for the result.
            url = new URL(mpsBaseUrl + mlInitialResponse.getRequestURL() + "/result?" + "format=p
            urlConnection = (HttpURLConnection) url.openConnection();
            urlConnection.setRequestProperty("Content-Type", CONTENT_TYPE);
            urlConnection.setRequestMethod("GET");
            urlConnection.connect();


            // Parse the response body of the above HTTP request using methods from MATLABResult
            // The MATLABParams object created earlier is an input argument to the newInstance me
            // If there is any error in MATLAB, call to getResult() throws a MATLABException whic
            // displayed in MATLAB.
            MATLABResult mlFinalResult1 = MATLABResult.newInstance(mlMakeBody, urlConnection.getI
            try{
                double[][] magicSq1 = (double[][]) mlFinalResult1.getResult();
                printResult(magicSq1);
            }catch(MATLABException e){
                e.printStackTrace();
            }
        } catch(Exception e){
            e.printStackTrace();
        }

    }

    // Helper method to print out the magic square generated by MATLAB based on the input.
    private static void printResult(double[][] result) {
        for (double[] row : result) {
            for (double element : row) {
                System.out.print(element + " ");
            }
            System.out.println();
        }
    }
}
```

See Also

More About

- “Synchronous RESTful Requests Using Protocol Buffers in the Java Client” on page 2-37
- “Struct Support for RESTful Requests Using Protocol Buffers in the Java Client” on page 2-41
- “Create Java Client Using MWHttpClient Class” on page 1-3
- “Create MATLAB Production Server Java Client Using MWHttpClient Class” on page 1-2

Synchronous RESTful Requests Using Protocol Buffers in the Java Client

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

To use protobuf when making a request to the server, set the HTTP `Content-Type` header to `application/x-google-protobuf` in the client code. The Java 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 Java client library provides methods and classes to deserialize the protobuf responses.

To use the Java client library, you must include `mps_client.jar` in the `CLASSPATH`.

The following table shows where to find the `mps_client.jar` file, Javadoc, and sample code for the example.

Location of `mps_client.jar`	• *`MPS_INSTALL`*`/client/java` • *`MATLABProductionServer_<release>_Clients`*`/java`
Location of Javadoc	• *`MPS_INSTALL`*`/client/java/doc` • *`MATLABProductionServer_<release>_Clients`*`/java/doc`
Location of code for the example files	• *`MPS_INSTALL`*`/client/java/examples` • *`MATLABProductionServer_<release>_Clients`*`/java/examples/MagicSquare`
• *`MPS_INSTALL`* is the location in which MATLAB Production Server is installed. • *`MATLABProductionServer_<release>_Clients`* is the folder containing MATLAB Production Server client libraries that you can download from https://www.mathworks.com/products/matlab-production-server/client-libraries.html.	

The example uses the `java.net` package for making HTTP requests to evaluate a MATLAB function deployed on a MATLAB Production Server instance running on `http://localhost:9910`.

Deploy your MATLAB Function on the Server

Write a MATLAB function `mymagic` that uses the `magic` function to create a magic square, then deploy it on the server.

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

```
function m = mymagic(in)

  m = magic(in);
end
```

The function `mymagic` takes a single `int32` input and returns a magic square as a 2-D `double` array.

Make a Synchronous Request to the Server

1. Construct the request URL.

   In the Java client, use the POST Synchronous Request to make the initial request to the server. The request URL comprises of the address of the server instance, the name of the deployed archive and the name of the MATLAB function to evaluate.

   ```
   String mpsBaseUrl = "http://localhost:9910";
   URL url;
   url = new URL(mpsBaseUrl + "/mymagic/mymagic");
   ```

2. Set the request headers.

   Set the HTTP `Content-Type` header to `application/x-google-protobuf`, as the API returns a byte array of protocol buffer messages.

   ```
   final static protected String CONTENT_TYPE = "application/x-google-protobuf";
   HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
   urlConnection.setDoOutput(true);
   urlConnection.setRequestProperty("Content-Type", CONTENT_TYPE);
   ```

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

   The function `mymagic` takes a single `int32` input and returns a magic square as a 2-D `double` array.

   Use the `newInstance(arg1, arg2, arg3)` method defined in the `MATLABParams` class to build the message. Since the `mymagic` function returns a single 2-D array, set `arg1` to `1` and `arg2` to `double[][].class`. Specify an integer value for `arg3`, which is the input to the `mymagic` function.

   ```
   MATLABParams mlMakeBody = MATLABParams.newInstance(1, double[][].class, 2);
   ```

4. Send the request to the server.

   Write the `MATLABParams` `mlMakeBody` object to the output stream of the HTTP request.

   ```
   OutputStream output = urlConnection.getOutputStream();
   output.write(mlMakeBody.getRequestBody());
   output.flush();
   ```

Receive and Interpret the Server Response

On successful execution of the HTTP 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` `mlMakeBody` object and the response body of the HTTP request to the `newInstance` method.

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

```
MATLABResult<double[][]> mlFinalResult1 =
    MATLABResult.newInstance(mlMakeBody, urlConnection.getInputStream());
try{
    double[][] magicSq1 = mlFinalResult1.getResult();
    printResult(magicSq1);
}catch(MATLABException e){
```

```
        e.printStackTrace();
    }
```

Write a helper method `printResult` which takes as input the result that is parsed from the response body and prints the corresponding 2-D array.

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

Sample code for the `SyncExample.java` Java client follows.

Code:

SyncExample.java

```
import com.mathworks.mps.client.MATLABException;
import com.mathworks.mps.client.rest.MATLABParams;
import com.mathworks.mps.client.rest.MATLABResult;

import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;


public class SyncExample{

    final static protected String CONTENT_TYPE = "application/x-google-protobuf";

    public static void main(String[] args){

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

            // Use the java.net package's URLConnection as HTTP Client in this example.
            URL url;
            url = new URL(mpsBaseUrl + "/mymagic/mymagic");
            HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
            urlConnection.setDoOutput(true);
            //Set Content-Type to protobuf.
            urlConnection.setRequestProperty("Content-Type", CONTENT_TYPE);

            // Make the initial POST request body with MATLABParams class.
            MATLABParams mlMakeBody = MATLABParams.newInstance(1, double[][].class, 2);

            // Write the MATLABParams object into the output stream of the HTTP request.
            OutputStream output = urlConnection.getOutputStream();
            output.write(mlMakeBody.getRequestBody());
            output.flush();

            // Parse the response body of the above HTTP request with methods from the MATLABResu
          // The MATLABParams object created earlier is an input argument to the newInstance meth
```

```
            // If there is any error in MATLAB, call to getResult() throws a MATLABException whic
            // displayed in MATLAB.
            MATLABResult mlFinalResult1 = MATLABResult.newInstance(mlMakeBody, urlConnection.getI
            try{
                double[][] magicSq1 = (double[][]) mlFinalResult1.getResult();
                printResult(magicSq1);
            }catch(MATLABException e){
                e.printStackTrace();
            }
        }catch(Exception e){
            e.printStackTrace();
        }
    }

    // Helper method to print out the magic square generated by MATLAB based on the input.
    private static void printResult(double[][] result) {
        for (double[] row : result) {
            for (double element : row) {
                System.out.print(element + " ");
            }
            System.out.println();
        }
    }
}
```

See Also

More About

- “Asynchronous RESTful Requests Using Protocol Buffers in the Java Client” on page 2-31
- “Struct Support for RESTful Requests Using Protocol Buffers in the Java Client” on page 2-41
- “Create Java Client Using MWHttpClient Class” on page 1-3
- “Create MATLAB Production Server Java Client Using MWHttpClient Class” on page 1-2

Struct Support for RESTful Requests Using Protocol Buffers in the Java Client

This example shows how to send MATLAB structures (`struct` (MATLAB)) represented as arrays of Java objects as input when you make a synchronous request using the Java 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.java`, for evaluating a MATLAB function deployed on the server.

To use protobuf when making a request to the server, set the HTTP `Content-Type` header to `application/x-google-protobuf` in the client code. The Java 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 Java client library provides methods and classes to deserialize the protobuf responses.

To use the Java client library, you must include `mps_client.jar` in the `CLASSPATH`.

The following table shows where to find the `mps_client.jar` file, Javadoc, and sample code for the example.

<table>
<tr><td>Location of <code>mps_client.jar</code></td><td>• MPS_INSTALL/client/java
• MATLABProductionServer_<release>_Clients/java</td></tr>
<tr><td>Location of Javadoc</td><td>• MPS_INSTALL/client/java/doc
• MATLABProductionServer_<release>_Clients/java/doc</td></tr>
<tr><td>Location of code for the example files</td><td>• MPS_INSTALL/client/java/examples
• MATLABProductionServer_<release>_Clients/java/examples/SortStudents</td></tr>
<tr><td colspan="2">• MPS_INSTALL is the location in which MATLAB Production Server is installed.
• MATLABProductionServer_<release>_Clients is the folder containing MATLAB Production Server client libraries that you can download from https://www.mathworks.com/products/matlab-production-server/client-libraries.html.</td></tr>
</table>

The example uses the `java.net` package for making HTTP requests to evaluate a MATLAB function deployed on a MATLAB Production Server instance running on `http://localhost:9910`.

Deploy your MATLAB function on the server

Write a MATLAB function `sortstudents` that takes an array of structures 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. For information on how to deploy, see "Create Deployable Archive for MATLAB Production Server".

```
function sorted = sortstudents(unsorted)

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

Create helper classes

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

```
class Student {
    String name;
    int score;
    String grade;
}
```

2 Create a Java class `StudentMarshaller` that extends the interface `MWDefaultMarshalingRules`. Since Java 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 and serialize Java objects to structs and deserialize structs to Java objects.

```
public class StudentMarshaller extends MWDefaultMarshalingRules {
    @override
    public List<Class> getStructTypes() {
    List structType = new ArrayList();
    structType.add(Student.class);
    return structType;
    }
}
```

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

```
Student[] students = new Student[]{new Student("Toni Miller", 90, "A"),
new Student("Ed Plum", 80, "B+"),
new Student("Mark Jones", 85, "A-")};
```

Make a synchronous request to the server

1 Construct the request URL.

In the Java client, use the POST Synchronous Request RESTful API to make the initial request to the server. The request URL comprises of the address of the server instance, the name of the deployed archive and the name of the MATLAB function to evaluate.

```
String mpsBaseUrl = "http://localhost:9910";
URL url;
url = new URL(mpsBaseUrl + "/sortstudents/sortstudents");
```

2 Set the request headers.

Set the HTTP `Content-Type` header to `application/x-google-protobuf`, as the API returns a byte array of protocol buffer messages.

```
final static protected String CONTENT_TYPE = "application/x-google-protobuf";
HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
urlConnection.setDoOutput(true);
urlConnection.setRequestProperty("Content-Type", CONTENT_TYPE);
```

3 Create the HTTP request body.

To create the HTTP request body, pass the `StudentMarshaller` class as an argument to the `MATLABParamsnewInstance` method. `StudentMarshaller` class serializes an array of Java objects of the class `Student` into an array of structs and deserializes the array of structs into to an array of Java objects of class `Student`.

```
MATLABParams mlMakeBody = MATLABParams.newInstance(1, Student[].class, new StudentMarshal
```

4 Send the request to the server.

   Write the `MATLABParams` `mlMakeBody` object to the output stream of the HTTP request.

```
OutputStream output = urlConnection.getOutputStream();
output.write(mlMakeBody.getRequestBody());
output.flush();
```

Receive and interpret the server response

On successful execution of the HTTP 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. Create a `MATLABResult` object using the `newInstance` method. The `newInstance` method takes the `MATLABParams` `mlMakeBody` object and the response body of the HTTP request as input arguments. Set the return type of the `MATLABResult` object to `Student[]`.

```
MATLABResult<Student[]> mlFinalResult = MATLABResult.newInstance(mlMakeBody, urlConnection.ge
try{
    Student[] magicSq = mlFinalResult.getResult();
    for (Student student : magicSq) {
    System.out.println(student);
    }
}catch(MATLABException e){
    e.printStackTrace();
}
```

Sample code for the `SortStudentsSyncREST.java` Java client and the `Student.java` helper class follows.

Code:

SortStudentsSyncREST.java

```
import com.mathworks.mps.client.MATLABException;
import com.mathworks.mps.client.MWDefaultMarshalingRules;
import com.mathworks.mps.client.rest.MATLABParams;
import com.mathworks.mps.client.rest.MATLABResult;

import java.io.IOException;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;


class StudentMarshaller extends MWDefaultMarshalingRules {
    @Override
    public List<Class> getStructTypes() {
        List structType = new ArrayList();
        structType.add(Student.class);
        return structType;
    }
}
```

```
public class SortStudentsSyncREST {
    final static protected String CONTENT_TYPE = "application/x-google-protobuf";

    public static void main(String[] args) {

        //Creating a Student Array
        Student[] students = new Student[]{new Student("Toni Miller", 90, "A"),
                new Student("Ed Plum", 80, "B+"),
                new Student("Mark Jones", 85, "A-")};

        // Use the java.net package's URLConnection as HTTP Client in this example
        try {
            String mpsBaseUrl = "http://localhost:9910";
            URL url;
            url = new URL(mpsBaseUrl + "/sortstudents/sortstudents");
            HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
            urlConnection.setDoOutput(true);
            //Content-Type needs to be set to protobuf
            urlConnection.setRequestProperty("Content-Type", CONTENT_TYPE);


            // This class makes the initial POST request body.
            MATLABParams mlMakeBody = MATLABParams.newInstance(1, Student[].class, new StudentMa

            // Write the MATLABParams object into the output stream of the HTTP request.
            OutputStream output = urlConnection.getOutputStream();
            output.write(mlMakeBody.getRequestBody());
            output.flush();

            // Parse the response body of the above HTTP request with the help of MATLABResult. 
            // here also takes MATLABParams object that was initially created.
            // If there is any error in MATLAB, call to getResult() will throw MATLABException wh
            // displayed in MATLAB.
            MATLABResult mlFinalResult = MATLABResult.newInstance(mlMakeBody, urlConnection.getIn
            Student[] magicSq = (Student[]) mlFinalResult.getResult();
            for (Student student :
                    magicSq) {
                System.out.println(student);
            }

        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

Student.java

```
class Student {
    String name;
    int score;
    String grade;

    public Student() {
    }
```

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

    public String getName() {
        return name;
    }

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

    public int getScore() {
        return score;
    }

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

    public String getGrade() {
        return grade;
    }

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

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", score=" + score +
                ", grade='" + grade + '\'' +
                '}';
    }
}
```

See Also

More About

- “Asynchronous RESTful Requests Using Protocol Buffers in the Java Client” on page 2-31
- “Synchronous RESTful Requests Using Protocol Buffers in the Java Client” on page 2-37
- “Create Java Client Using MWHttpClient Class” on page 1-3
- “Create MATLAB Production Server Java Client Using MWHttpClient Class” on page 1-2

Evaluate Deployed Machine Learning Models Using Java Client

This example shows how to write a client application that uses the MATLAB Production Server Java client library to evaluate a machine learning model deployed to MATLAB Production Server. The **Classification Learner** app is used to train and export the model in this example. Typically, a MATLAB developer trains a model and exports the trained model as a deployable archive (CTF file) using either the **Classification Learner** app or **Regression Learner** app. For details, see “Deploy Model Trained in Classification Learner to MATLAB Production Server” (Statistics and Machine Learning Toolbox) and “Deploy Model Trained in Regression Learner to MATLAB Production Server” (Statistics and Machine Learning Toolbox). The Statistics and Machine Learning Toolbox™ is required to use **Classification Learner** and **Regression Learner**. A server administrator deploys the archive to a MATLAB Production Server instance.

The example provides and explains how to use a sample Java client, `PredictFunctionPatientData.java`, for sending patient data that is in `patients.csv` to a MATLAB function `predictFunction` deployed on the server. The result of `predictFunction` classifies the patient data as a smoker or nonsmoker. The example also uses helper classes `PatientData.java`, `PatientDataBeanInfo.java`, and `Utils.java`.

The files in the example are available online at MATLAB Production Server Client Libraries. In an on-premises MATLAB Production Server installation, the example files are located in *`$MPS_INSTALL`*`/client/java/examples/ClassificationModelPatientData`, where *`$MPS_INSTALL`* is the MATLAB Production Server installation location. The `examples` directory also contains a sample Java client to evaluate a deployed machine learning model created using the **Regression Learner** app. For an overview of how to write a client using the Java client library, see “Create MATLAB Production Server Java Client Using MWHttpClient Class” on page 1-2.

Determine Type of Input Argument for Deployed Function

In the scenario for this example, the MATLAB developer determines whether the deployed MATLAB model requires input data as a matrix or a table.

If the model requires a table, Java client programs must send an array of objects instead, since the MATLAB Production Server Java client library does not support the `table` data type. When used as an input to a MATLAB function, Java objects get marshaled into MATLAB `struct` (MATLAB), since Java does not natively support MATLAB structures.

In this example, the deployed `predictFunction` MATLAB function requires the input in the form of an array of objects that get marshaled as `structs`.

Represent Input Data as Array of Objects

The file `patients.csv` contains the input patient data. Each row in `patients.csv` corresponds to one patient, and the comma separated values in each row correspond to a diagnostic variable. The `Smoker` variable is the response variable, and the rest of the variables—`Age`, `Diastolic`, `Gender`, `Height`, `SelfAssessedHealthStatus`, `Systolic`, `Weight`—are predictors.

The following sections explain how to convert the patient data from the CSV file into an array of objects. The deployed MATLAB function `predictFunction` requires the patient data input to be an array of objects.

Create Java Class to Represent Input Data

Create a Java class `PatientData` to represent data from the `patients.csv` file. Each object of the `PatientData` class represents a single row in `patients.csv`. The instance variable names in `PatientData` must be the same as the predictor variable names in `patients.csv`. The predictor variable names start with an uppercase letter. However, the instance variable names start with a lowercase letter, by default. Later, you see how to override this default behavior to match the casing of the predictor variable names and instance variable names.

The class definition for `PatientData.java` follows.

```
package com.mathworks;

public class PatientData {
    double age;
    double diastolic;
    String gender;
    double height;
    String selfAssessedHealthStatus;
    double systolic;
    double weight;

    public PatientData(double age, double diastolic, String gender, double height, String selfAss
        this.age = age;
        this.diastolic = diastolic;
        this.gender = gender;
        this.height = height;
        this.selfAssessedHealthStatus = selfAssessedHealthStatus;
        this.systolic = systolic;
        this.weight = weight;
    }

    public double getAge() {
        return age;
    }

    public void setAge(double age) {
        this.age = age;
    }

    public double getDiastolic() {
        return diastolic;
    }

    public void setDiastolic(double diastolic) {
        this.diastolic = diastolic;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public double getHeight() {
```

```
        return height;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    public String getSelfAssessedHealthStatus() {
        return selfAssessedHealthStatus;
    }

    public void setSelfAssessedHealthStatus(String selfAssessedHealthStatus) {
        this.selfAssessedHealthStatus = selfAssessedHealthStatus;
    }

    public double getSystolic() {
        return systolic;
    }

    public void setSystolic(double systolic) {
        this.systolic = systolic;
    }

    public double getWeight() {
        return weight;
    }

    public void setWeight(double weight) {
        this.weight = weight;
    }

    @Override
    public String toString() {
        return "PatientData{" +
                "age=" + age +
                ", diastolic=" + diastolic +
                ", gender='" + gender + '\'' +
                ", height=" + height +
                ", selfAssessedHealthStatus='" + selfAssessedHealthStatus + '\'' +
                ", systolic=" + systolic +
                ", weight=" + weight +
                '}';
    }
}
```

Map Java Instance Variables to MATLAB Structure Members

Java classes that represent MATLAB structures use the Java Beans `Introspector` class to map instance variables to fields of the structure. For more information about the `Introspector` class, see the Oracle Documentation for Class Introspector. These classes also use the default naming conventions of the `Introspector` class. The default convention uses the `decapitalize` method that maps the first letter of a Java variable name into a lowercase letter. Therefore, using the default, you cannot define a Java instance variable that maps to a MATLAB structure member that starts with an uppercase letter. You can override this behaviour by implementing a `SimpleBeanInfo` class with a custom `getPropertyDescriptors()` method.

The example uses a `PatientDataBeanInfo` class that implements the `SimpleBeanInfo` interface and sets the first letter of the instance variables of `PatientDataBeanInfo` to uppercase. The code for `PatientDataBeanInfo` class follows.

```
package com.mathworks;

import java.beans.IntrospectionException;
import java.beans.PropertyDescriptor;
import java.beans.SimpleBeanInfo;

public class PatientDataBeanInfo extends SimpleBeanInfo
{
    @Override
    public PropertyDescriptor[] getPropertyDescriptors()
    {
        PropertyDescriptor[] props = new  PropertyDescriptor[7];
        try
        {
            props[0] = new PropertyDescriptor("Age",PatientData.class,
                    "getAge","setAge");
            props[1] = new PropertyDescriptor("Diastolic",PatientData.class,
                    "getDiastolic","setDiastolic");
            props[2] = new PropertyDescriptor("Gender",PatientData.class,
                    "getGender","setGender");
            props[3] = new PropertyDescriptor("Height",PatientData.class,
                    "getHeight","setHeight");
            props[4] = new PropertyDescriptor("SelfAssessedHealthStatus",PatientData.class,
                    "getSelfAssessedHealthStatus","setSelfAssessedHealthStatus");
            props[5] = new PropertyDescriptor("Systolic",PatientData.class,
                    "getSystolic","setSystolic");
            props[6] = new PropertyDescriptor("Weight",PatientData.class,
                    "getWeight","setWeight");

            return props;
        }
        catch (IntrospectionException e)
        {
            e.printStackTrace();
        }

        return null;
    }
}
```

Prepare Data for Input to Deployed Function

The example provides a class `Utils.java` that contains utility functions that convert the data from the `patients.csv` file to an array of objects as expected by the deployed `predictFunction` MATLAB function. `Utils.java` contains a function that reads the data from the CSV file and converts it into an `ArrayList` class. `Utils.java` contains another function that converts the `ArrayList` class into an array of `PatientData` objects.

Code for `Utils.java` follows.

```
package com.mathworks;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;
import java.util.StringTokenizer;

public class Utils {
    String fileName;
```

```
    Utils(String fileName) {
        this.fileName = fileName;
    }

    public static ArrayList<String[]> readFromCsv() {
        Scanner sc = null;
        ArrayList<String[]> inputs = new ArrayList<String[]>();

        try {
            sc = new Scanner(new File("patients.csv"));

            sc.useDelimiter("\n");
            sc.next();
            String line;

            while (sc.hasNext())
            {
                line = sc.next();
                StringTokenizer tokenizer = new StringTokenizer(line, ",");
                String[] tmp = new String[tokenizer.countTokens() - 1];
                for (int i = 0; i < tmp.length; i++) {
                    String s = tokenizer.nextToken();
                    if (s.compareTo("Inf") == 0) {
                        tmp[i] = String.valueOf(Double.POSITIVE_INFINITY);
                    } else if (s.compareTo("-Inf") == 0) {
                        tmp[i] = String.valueOf(Double.NEGATIVE_INFINITY);
                    } else {
                        tmp[i] = s;
                    }
                }
                inputs.add(tmp);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }finally{
            sc.close();
        }

        return inputs;
    }

    public static PatientData[] prepareDataForTableInput() {
        ArrayList<String[]> inputs = readFromCsv();
        int i = 0;
        PatientData[] datas = new PatientData[inputs.size()];
        for (String[] mat : inputs) {
            datas[i] = new PatientData(Double.valueOf(mat[0]),Double.valueOf(mat[1]),mat[2],Doub
            i++;
        }

        return datas;
    }
}
```

Write Client Application

In the client application, define a Java interface that represents the deployed MATLAB function. Then, instantiate a proxy object to communicate with the server and call the deployed function.

Since the deployed function expects MATLAB structures as input, you must use the `MWStructureList` annotation for the interface to include the `PatientData` class that you created earlier. For an additional example on how to marshal MATLAB structures in Java, see “Marshal MATLAB Structures (Structs) in Java” on page 2-18. For an example on instantiating a proxy object and calling deployed functions, see “Create Java Client Using MWHttpClient Class”.

The code for the client application `PredictFunctionPatientData.java` follows.

```
package com.mathworks;

import com.mathworks.mps.client.MATLABException;
import com.mathworks.mps.client.MWClient;
import com.mathworks.mps.client.MWHttpClient;
import com.mathworks.mps.client.annotations.MWStructureList;

import java.io.IOException;
import java.net.URL;

interface CallMethod {
    @MWStructureList({PatientData.class})
    boolean[] predictFunction(PatientData[] dataSet) throws IOException, MATLABException;
}

public class PredictFunctionPatientData {

    public static void main(String[] args) {
        PatientData[] datas = Utils.prepareDataForTableInput();

        MWClient client = new MWHttpClient();
        try {
            CallMethod s = client.createProxy(new URL("http://localhost:9910/DeployedClassificationModel"), CallMethod.cl

            boolean[] results = s.predictFunction(datas);
            for (int j = 0; j < results.length; j++) {
                System.out.println(results[j]);
            }
        } catch (MATLABException ex) {
            System.out.println(ex);
        } catch (IOException ex) {
            System.out.println(ex);
        } finally {
            client.close();
        }
    }
}
```

See Also

Related Examples

- “Deploy Model Trained in Classification Learner to MATLAB Production Server” (Statistics and Machine Learning Toolbox)
- “Deploy Model Trained in Regression Learner to MATLAB Production Server” (Statistics and Machine Learning Toolbox)
- “Marshal MATLAB Structures (Structs) in Java” on page 2-18
- “Create Java Client Using MWHttpClient Class”

- “Create MATLAB Production Server Java Client Using MWHttpClient Class” on page 1-2

External Websites

- Oracle Documentation for Class Introspector

Security

- "Execute MATLAB Functions Using HTTPS" on page 3-2
- "Customize Security Configuration" on page 3-8

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 using a Java client:

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 the MATLAB Production Server Java client library, see “Create Java Client Using MWHttpClient Class” on page 1-3.

The MATLAB Production Server Java client API provides hooks for the following:

- Disabling security protocols to protect against the POODLE vulnerability.
- Providing your own `HostnameVerifier` implementation.
- Implementing server authorization beyond that provided by HTTPS.

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 Java trust store on the client machine. If the server uses a self-signed certificate or if the root certificate of the server signed by a certificate authority (CA) is not present in the Java trust store, obtain the server certificate from the MATLAB Production Server administrator or export the certificate using a browser, then import the server certificate into the Java trust 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**. This opens a wizard that lets you export the SSL certificate. Click **Next**.
4. You can use the default format selection of `DER encoded binary X.509 (.CER)`. 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 Java Trust Store

The default Java trust store is located in `${JAVA_HOME}\lib\security\cacerts`. You can use the `keytool` utility located in `${JDK_HOME}\bin` to import the SSL certificate of the server into the trust store on the client machine. For more information, see keytool.

Import the server certificate to the Java trust store of the client machine using the following command:

```
C:\tmp>keytool -importcert -file PATH_TO_SERVER_CERTIFICATE\server_cert.cer -keystore client.trus
```

Doing so imports the server certificate `server_cert.cer` into a trust store and generates a `client.truststore` file in the current working directory. You can specify `client.truststore` file as the trust store when you write the client program to establish a secure proxy connection.

To use a location other than the default for the client trust store, set the trust store location and password using Java system properties, either using Java code or when running you Java client program the command line.

- Set Java system properties in your code:

  ```
  System.setProperty("javax.net.ssl.trustStore",
                     "PATH_TO_TRUSTSTORE\\client.truststore");
  System.setProperty("javax.net.ssl.trustStorePassword",
                     "TRUSTSTORE_PASSWORD");
  MWClient client = new MWHttpClient();
  URL sslURL = new URL("https://server FQDN:port/myApplication");

  MyProxy sslProxy = client.createProxy(sslURL, MyProxy.class);
  ```
- Set Java system properties using the command line at run time:

  ```
  C:\>java -Djavax.net.ssl.trustStore="client.truststore" -Djavax.net.ssl.trustStorePassword="TR
  ```

To connect to a server that requires client-side authentication, a client certificate must also be present in the key store of the client. For more information, see “Establish Secure Connection Using Client Authentication” on page 3-3.

Establish Secure Proxy Connection

After your client machine is configured to use the server certificate or if the server uses a CA-signed SSL certificate that Java trusts, you can write your client program to create a secure proxy connection with the server using the following code:

```
MWClient client = new MWHttpClient();
URL sslURL = new URL("https://server FQDN:port/myApplication");
MyProxy sslProxy = client.createProxy(sslURL, MyProxy.class);
```

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 default Java trust store of the client machine to perform the HTTPS server authentication. If the server requests client authentication, the HTTPS handshake fails because the default `SSLContext` object created by the JRE does not provide a key store.

Establish Secure Connection Using Client Authentication

Before a .NET client can communicate with a server instance that requires client authentication, you must create a client certificate.

Create Client Certificate

1. On the client machine, create a client certificate in JKS format in the key store.

```
C:\tmp>keytool -genkey -alias javaclient -keystore client.jks
```

The command creates a certificate `client.jks` with an alias `javaclient`.

2 In your client program, set the key store location using the file `client.jks` and password using Java system properties.

```
System.setProperty("javax.net.ssl.keyStore", "PATH_TO_KEYSTORE\\client.jks");
System.setProperty("javax.net.ssl.keyStorePassword", "keystore_pass");
MWClient client = new MWHttpClient();
URL sslURL = new URL("https://hostname:port/myApplication");
MyProxy sslProxy = client.createProxy(sslURL, MyProxy.class);
```

Save Client Certificate on Server

1 Export the public client certificate `client.jks` in DER format using `keytool`, then transform it to PEM format using `openssl`.

```
C:\tmp>keytool -export -keystore client.jks -alias javaclient -file client.cer
C:\tmp>openssl x509 -inform DER -in client.cer -outform PEM -text -out client_cert.pem
```

2 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”.

Handle Exceptions

Override Certificate Check

If the self-signed certificate or the root CA certificate of the server is not present in the Java trust 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:

```
javax.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException:
PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException:
unable to find valid certification path to requested target
```

Use one of the following options to handle this exception:

- Add the SSL certificate of the server to the Java trust store on the client machine. For more information, see “Add Certificate to Java Trust Store” on page 3-2.
- Override the certificate check and accept the untrusted certificate using the following code. The code provides a custom implementation of the `MWSSLConfig` interface to use a custom `SSLContext` implementation.

```
MWSSLConfig sslConfig = new MWSSLDefaultConfig(){
    public SSLContext getSSLContext(){
        try {
            TrustManager[] trustAllCerts = new TrustManager[] {
                new X509TrustManager() {
                    public java.security.cert.X509Certificate[] getAcceptedIssuers() {
                        return null;
                    }

                    public void checkClientTrusted(X509Certificate[] certs, String authType) {
```

```
                    public void checkServerTrusted(X509Certificate[] certs, String authType) {
                }
            };

            SSLContext sc = SSLContext.getInstance("SSL");
            sc.init(null, trustAllCerts, new java.security.SecureRandom());
            return sc;
        } catch(Exception ex){
            throw new RuntimeException("Error creating SSLContext : ", ex);
        }
    }
};

// Create a non-interruptible MWHttpClient instance
final MWClient client = new MWHttpClient(sslConfig);
```

This option is not recommended for a production environment.

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:

```
class MySSLConfig extends MWSSLDefaultConfig {
  public HostnameVerifier getHostnameVerifier() {
    return new HostnameVerifier() {
      public boolean verify(String s, SSLSession sslSession) {
        return true;
      }
    };
  }
}
```

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)”.

Sample Code

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

MagicAsync.java

```
import java.net.URL;
import java.util.concurrent.Future;
import com.mathworks.mps.client.*;
import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.SSLSession;
```

```
class MyConfig extends MWHttpClientDefaultConfig{
  public boolean isInterruptible() { return true; }
  public int getMaxConnectionsPerAddress() { return 10; }
}

class MySSLConfig extends MWSSLDefaultConfig {
  public HostnameVerifier getHostnameVerifier() {
    return new HostnameVerifier() {
      public boolean verify(String s, SSLSession sslSession) {
        return true;
      }
    };
  }
}

public class MagicAsync{
  public static void main(String[] args){
    MWClient client = new MWHttpClient( new MyConfig(), new MySSLConfig() );

    try{
      MWInvokable invokable = client.createComponentProxy(new URL("https://localhost:9920/mymagi

      MWInvokeRequest<double[][]> httpRequest = new MWInvokeRequest("mymagic", double[][].class)
      httpRequest.setInputParams(4);
      httpRequest.setNargout(1);

      MWRequest<double[][]> request = invokable.invokeAsync(httpRequest, null);
      Future<double[][]> f = request.getFuture();

      double[][] res = f.get();
      printResult(res);
    }
    catch(Exception ex){
      System.out.println(ex);
    }
    finally{
      client.close();
    }
  }

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

See Also

More About

- "Customize Security Configuration" on page 3-8

- "Enable HTTPS"
- "Create Java Client Using MWHttpClient Class" on page 1-3

External Websites

- keytool

Customize Security Configuration

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

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

You extend the MWSSLDefaultConfig class to:

- specify the security protocols the client can use
- customize how the client verifies hostnames
- specify additional server authentication logic

The MWSSLDefaultConfig class has three methods:

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

Specify Enabled Encryption Protocols

MATLAB Production Server supports the following encryption protocols:

- TLSv1.0
- TLSv1.1
- TLSv1.2

By default, all protocols are enabled. If you want to control which protocols are enabled, you override the getSSLContext() method to return an instance of MWCustomSSLContext with a list of enabled protocols. Protocols not on the list are not enabled. For example, to avoid the POODLE vulnerability by disabling SSL protocols, you enable the TLS protocols.

```
import javax.net.ssl.SSLContext;
import java.security.KeyManagementException;
import java.security.NoSuchAlgorithmException;
import com.mathworks.mps.client.*;

public class MySSLConfig extends MWSSLDefaultConfig
{
  public SSLContext getSSLContext()
  {
    try
    {
      final SSLContext context = MWCustomSSLContext.getInstance("TLSv1", "TLSv1.1", "TLSv1.2");
```

```
      context.init(null,null,null);
      return context;
    }
    catch (NoSuchAlgorithmException e)
    {
      return null;
    }
    catch (KeyManagementException e)
    {
      return null;
    }
  }
}
```

Override Default Hostname Verification

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

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

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

```
import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.SSLSession;
import com.mathworks.mps.client.*;

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

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

For information on disabling host name verification, see “Disable Host Name Verification” on page 3-5.

Use Additional Server Authentication

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

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

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

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

See Also

More About

- “Execute MATLAB Functions Using HTTPS” on page 3-2

Data Conversion Rules

Conversion of Java Types to MATLAB Types

Value Passed to Java Method is:	Input type Received by MATLAB is:	Dimension of Data in MATLAB is:
`java.lang.Byte`, `byte`	`int8`	{1,1}
`byte[]` *data*		{1, *data.length*}
`java.lang.Shortshort`	`int16`	{1,1}
`short[]` *data*		{1, *data.length*}
`java.lang.Integer`, `int`	`int32`	{1,1}
`int[]` *data*		{1, *data.length*}
`java.lang.Long`, `long`	`int64`	{1,1}
`long[]` *data*		{1, *data.length*}
`java.lang.Float`,`float`	`single`	{1,1}
`float[]` *data*		{1, *data.length*}
`java.lang.Double`, `double`	`double`	{1,1}
`double[]` *data*		{1, *data.length*}
`java.lang.Boolean`, `boolean`	`logical`	{1,1}
`boolean[]` *data*		{1, *data.length*}
`java.lang.Character`, `char`	`char`	{1,1}
`char[]` *data*		{1, *data.length*}
`java.lang.String` *data*		{1, *data.length*()}
`java.lang.String[]` *data*	`cell`	{1, *data.length*}
`java.lang.Object[]` *data*		{1, *data.length*}
`T[]` *data*	MATLAB type for `T`	{ *data.length*, *dimensions*(`T[0]`) }, if `T` is an array
		{ 1, *data.length*}, if `T` is not an array

Conversion of MATLAB Types to Java Types

When MATLAB Returns:	Dimension of Data in MATLAB is:	MATLAB Data Converts To Java Type:
int8, uint8	{1,1}	byte,java.lang.Byte
	{1,*n*} , {*n*,1}	byte[*n*], java.lang.Byte[*n*]
	{*m*,*n*,*p*,...}	byte[*m*][*n*][*p*]... , java.lang.Byte[*m*][*n*][*p*]...
int16, uint16	{1,1}	short, java.lang.Short
	{1,*n*} , {*n*,1}	short[*n*], java.lang.Short[*n*]
	{*m*,*n*,*p*,...}	short[*m*][*n*][*p*]... , java.lang.Short[*m*][*n*][*p*]...
int32, uint32	{1,1}	int, java.lang.Integer
	{1,*n*} , {*n*,1}	int[*n*], java.lang.Integer[*n*]
	{*m*,*n*,*p*,...}	int[*m*][*n*][*p*]... , java.lang.Integer[*m*][*n*][*p*]...
int64, uint64	{1,1}	long, java.lang.Long
	{1,*n*} , {*n*,1}	long[*n*], java.lang.Long[*n*]
	{*m*,*n*,*p*,...}	long[*m*][*n*][*p*]... , java.lang.Long[*m*][*n*][*p*]...
single	{1,1}	float, java.lang.Float
	{1,*n*} , {*n*,1}	float[*n*], java.lang.Float[*n*]
	{*m*,*n*,*p*,...}	float[*m*][*n*][*p*]... , java.lang.Float[*m*][*n*][*p*]...
double	{1,1}	double, java.lang.Double
	{1,*n*} , {*n*,1}	double[*n*], java.lang.Double[*n*]
	{*m*,*n*,*p*,...}	double[*m*][*n*][*p*]... , java.lang.Double[*m*][*n*][*p*]...
logical	{1,1}	boolean, java.lang.Boolean
	{1,*n*} , {*n*,1}	boolean[*n*], java.lang.Boolean[*n*]
	{*m*,*n*,*p*,...}	boolean[*m*][*n*][*p*]... , java.lang.Boolean[*m*][*n*][*p*]...
char	{1,1}	char, java.lang.Character
	{1,*n*} , {*n*,1}	java.lang.String
	{*m*,*n*,*p*,...}	char[*m*][*n*][*p*]... , java.lang.Character[*m*][*n*][*p*]...
cell (containing only strings)	{1,1}	java.lang.String
	{1,*n*} , {*n*,1}	java.lang.String[*n*]
	{*m*,*n*,*p*,...}	java.lang.String[*m*][*n*][*p*]...

When MATLAB Returns:	Dimension of Data in MATLAB is:	MATLAB Data Converts To Java Type:
`cell` (containing multiple types)	{1,1}	`java.lang.Object`
	{1,*n*} , {*n*,1}	`java.lang.Object`[*n*]
	{*m*,*n*,*p*,...}	`java.lang.Object`[*m*][*n*][*p*]...