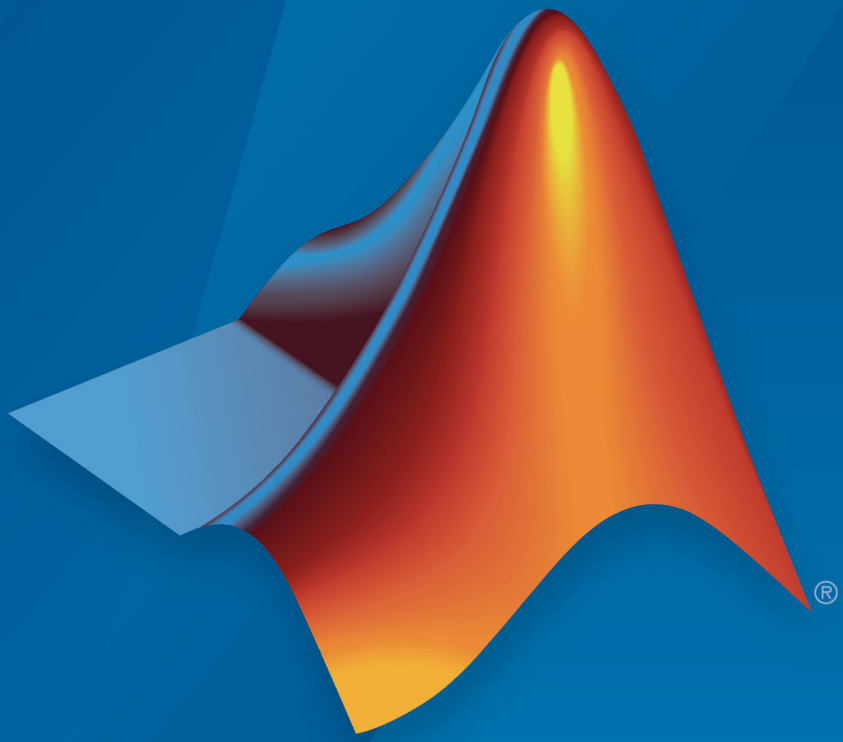
MATLAB® Production Server™

RESTful API and JSON



MATLAB®

R2017b

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

MATLAB® Production Server™ RESTful API and JSON

Revision History

March 2016	Online only	New for Version 2.3 (Release R2016a)
September 2016	Online only	Revised for Version 2.4 (Release R2016b)
March 2017	Online only	Revised for Version 3.0 (Release 2017a)
September 2017	Online only	Revised for Version 3.0.1 (Release R2017b)

Contents

1 Client Programming

RESTful API **1-2**
Synchronous Execution **1-2**
Example: Synchronous Execution of Magic Square using RESTful API and JSON **1-4**
Asynchronous Execution **1-6**
Example: Asynchronous Execution of Magic Square using RESTful API and JSON **1-8**

2 JSON Representation of MATLAB Data Types

JSON Representation of MATLAB Data Types **2-2**
Numeric Types: double, single **2-4**
Numeric Types: NaN, Inf, -Inf **2-4**
Numeric Types: Integers **2-6**
Numeric Types: Complex Numbers **2-6**
Characters **2-7**
Logical **2-8**
Cell Arrays **2-8**
Structures **2-9**
Empty Arrays: [] **2-10**
Multidimensional Arrays **2-10**

3 Troubleshooting RESTful API Errors

Troubleshooting RESTful API Errors **3-2**
Structure of MATLAB Error **3-4**
Access-Control-Allow-Origin **3-5**

4 Examples: RESTful API and JSON

Example: Web-Based Bond Pricing Tool Using JavaScript . . **4-2**
Step 1: Write MATLAB Code **4-2**
Step 2: Create a Deployable Archive with the Production Server Compiler App **4-3**
Step 3: Place the Deployable Archive on a Server **4-3**
Step 4: Enable Cross-Origin Resource Sharing (CORS) on the Server **4-3**
Step 5: Write JavaScript Code using the RESTful API and JSON **4-4**
Step 6: Embed JavaScript within HTML Code **4-5**
Step 7: Run Example **4-7**

5 RESTful API Reference

1

Client Programming

RESTful API

In this section...
“Synchronous Execution” on page 1-2 “Example: Synchronous Execution of Magic Square using RESTful API and JSON” on page 1-4 “Asynchronous Execution” on page 1-6 “Example: Asynchronous Execution of Magic Square using RESTful API and JSON” on page 1-8

The RESTful API uses the request-response model of the Hypertext Transfer Protocol (HTTP) for communication with MATLAB Production Server. This model includes request methods, response codes, message headers, and message bodies. The RESTful API has the following characteristics:

- The HTTP methods—POST, GET, and DELETE—form the primary mode of communication between client and server.
- Resources created by the server are uniquely identified using Uniform Resource Identifiers (URIs).
- Metadata such as the Content-type of a request is conveyed through a message header. The only supported Content-type is `application/json`.
- Inputs to the MATLAB function contained within a deployed archive are represented in JSON and encapsulated within the body of a message.
- The message body of the response contains information about a request such as state or results.
- Support for both the synchronous and asynchronous modes of the server.

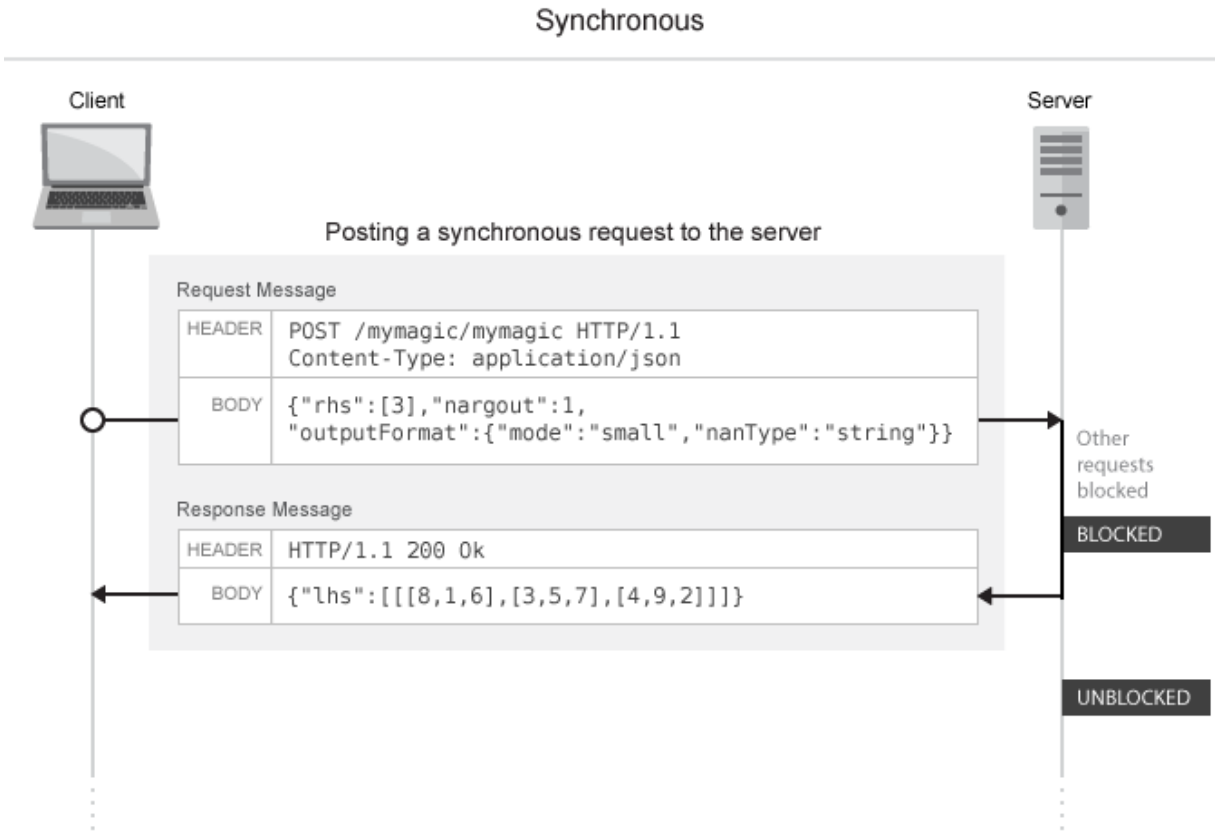
Synchronous Execution

In synchronous mode, once a client posts a request, the server blocks all further requests until it has completed processing the original request. Once processing is complete, the server automatically returns a response to the client.

RESTful API Calls for Synchronous Mode

Call	Purpose
POST Synchronous Request	Make a synchronous request to the server, and wait for a response

The following graphic illustrates how the RESTful API works in synchronous mode.



Example: Synchronous Execution of Magic Square using RESTful API and JSON

This example shows how to use the RESTful API and JSON by providing two separate implementations—one using JavaScript on page 1-4 and the other using Python on page 1-5. When you execute this example, the server returns a list of twenty-five comma-separated values. These values are the output of the deployed MATLAB function `mymagic`, represented in column-major format. The MATLAB code for the `mymagic` function follows.

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

For this example to run, a MATLAB Production Server instance containing the deployed MATLAB function `mymagic` needs to be running. For more information on how to create a deployable archive, see “Create a Deployable Archive for MATLAB Production Server”. For more information on setting up a server, see “Create a Server”.

JavaScript Implementation

With the JavaScript implementation of the RESTful API, you include the script within the `<script> </script>` tags of an HTML page. When this HTML page is opened in a web browser, the values of the `mymagic` function are returned. Note that the server needs to have `CORS` enabled for JavaScript code to work. For more information, see `cors-allowed-origins`.

Code:

restApiSyncMagicJavaScript.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>Magic Square</title>
        <script>
            var request = new XMLHttpRequest();
            //MPS RESTful API: Specify URL
            var url = "http://localhost:9910/ctfArchiveName/mymagic";
            //MPS RESTful API: Specify HTTP POST method
            request.open("POST",url);
            //MPS RESTful API: Specify Content-Type to application/json
            request.setRequestHeader("Content-Type", "application/json");
```

```
            var params = { "nargout": 1,
                           "rhs": [5] };
            request.send(JSON.stringify(params));
            request.onreadystatechange = function() {
                 if(request.readyState == 4)
                 {     //MPS RESTful API: Check for HTTP Status Code 200
                    if(request.status == 200)
                    {      result = JSON.parse(request.responseText);
                       if(result.hasOwnProperty("lhs")) {
                       //MPS RESTful API: Index into "lhs" to retrieve response from se
                       document.getElementById("demo").innerHTML = '<p>' + result.lhs[0
                    }
                       else if(result.hasOwnProperty("error")) {
                       alert("Error: " + result.error.message); }
                 }
            };
        </script>
    </head>
    <body>
        <p>MPS RESTful API and JSON EXAMPLE</p>
        <p> >> mymagic(5)</p>
        <p id="demo"></p>
        <p> # output from server returned in column-major format </p>
    </body>
</html>
```

Python Implementation

This examples uses Python 2.x. If you are using Python 3.x, you need to change some portions of the code.

Code:

restApiSyncMagicPython.py

```
#!/usr/bin/python
#This example uses Python 2.x
#In Python 3.x use:
#import http.client
#conn = http.client.HTTPConnection("localhost:9910")

import httplib
import json

conn = httplib.HTTPConnection("localhost:9910")
```

```
headers = { "Content-Type": "application/json"}
body = json.dumps({"nargout": 1, "rhs" : [5]})
conn.request("POST", "/ctfArchiveName/mymagic", body, headers)
response = conn.getresponse()
if response.status == 200:
   result = json.loads(response.read())
   if "lhs" in result:
     print("Result of magic(5) is " + str(result["lhs"][0]["mwdata"]))
   elif "error" in result:
     print("Error: " + str(result["error"]["message"]))
```

To learn how to deploy a MATLAB function on MATLAB Production Server and invoke it using RESTful API and JSON, see “Example: Web-Based Bond Pricing Tool Using JavaScript” on page 4-2.

Asynchronous Execution

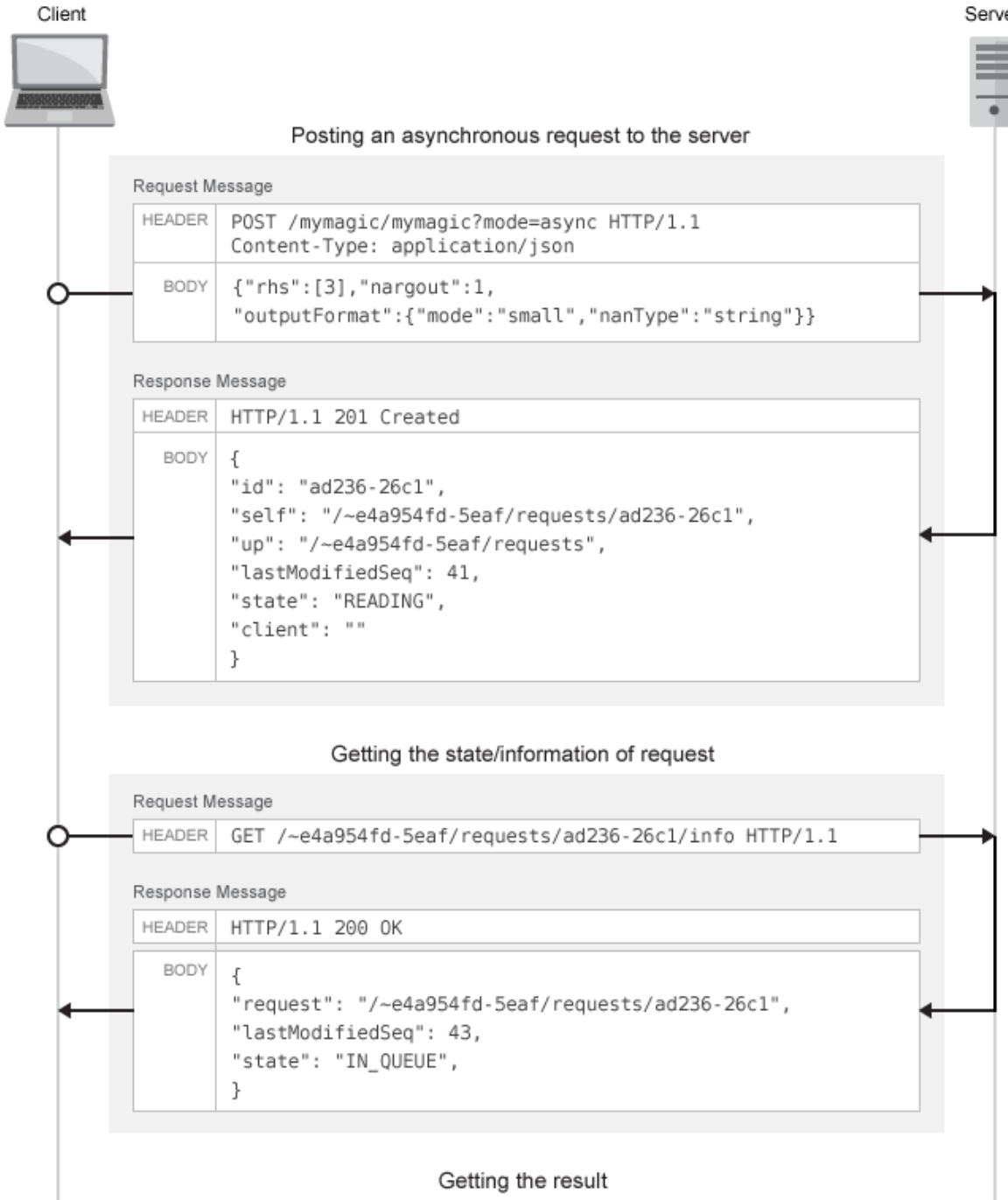
In asynchronous mode, a client can post multiple requests, and in each case the server responds by creating a new resource and returning a unique URI corresponding to each request. The URI is encapsulated within the body of the response message. The client can use the URI returned by the server for the purposes of querying and retrieving results among other uses.

RESTful API Calls for Asynchronous Mode

Call	Purpose
POST Asynchronous Request	Make an asynchronous request to the server
GET Representation of Asynchronous Request	View how an asynchronous request made to the server is represented
GET Collection of Requests	View a collection of requests
GET State Information	Get state information of a request
GET Result of Request	Retrieve the results of a request
POST Cancel Request	Cancel a request
DELETE Request	Delete a request

The following graphic illustrates how the RESTful API works in asynchronous mode. The graphic does not cover all the RESTful API calls. For a complete list of calls, see the preceding table.

Asynchronous



Example: Asynchronous Execution of Magic Square using RESTful API and JSON

This example shows how to use the RESTful API and JSON for asynchronous execution using JavaScript. When you execute this example, the server returns a list of one-hundred comma-separated values. These values are the output of the deployed MATLAB function `mymagic`, represented in column-major format. The MATLAB code for the `mymagic` function follows.

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

For this example to run, a MATLAB Production Server instance containing the deployed MATLAB function `mymagic` needs to be running. For more information on how to create a deployable archive, see “Create a Deployable Archive for MATLAB Production Server”. For more information on setting up a server, see “Create a Server”.

Code:

restApiAsyncMagicJavaScript.html

```
<!DOCTYPE html>
<html>

    <head>
        <title>Magic Square</title>
        <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js">
        <script>
            // MPS RESTful API (Asynchronous): Specify URL
            var hostname = "http://localhost:9910";
            var mode = "async";
            var clientID = "client100";
            var ctfName = "mymagic";
            var matlabFuncName = "mymagic"
            var url = hostname + "/" + ctfName + "/" + matlabFuncName + "?mode=" + mode
            // Specify arguments
            var params = {
                "nargout": 1,
                "rhs": [100],
                "outputFormat": {"mode": "small"}
            };
            $.ajax(url, {
                data: JSON.stringify(params),
```

```
    //MPS RESTful API (Asynchronous): Specify Content-Type to application/j
    contentType: 'application/json',
    method: 'POST',
    dataType: 'json',
    success: function(response) {
        // Print Request URI to webpage
        $("#requestURI").html('<strong>Request URI: </strong>' + hostname +
        pollUsingUp(response);
    }
});
// Polling Server using UP
function pollUsingUp(request) {
    setTimeout(function() {
        var newSeq = parseInt(request.lastModifiedSeq) + 1;
        var queryURI = hostname + request.up + "?since=" + newSeq + "&ids="
        $.ajax({
            url: queryURI,
            method: 'GET',
            dataType: 'json',
            success: function(response) {
                //Poll again if no data about the request was received.
                if (response.data.length == 0) {
                    pollUsingUp(request);
                    return;
                }

                var requestResource = response.data[0];
                // Print "state" of request
                $("#state").html('<strong>State: </strong>' + requestResour


                if (requestResource.state != "READY" && requestResource.sta
                    //Keep polling if the request is not done yet.
                    pollUsingUp(requestResource);
                } else {
                    var requestURI = hostname + requestResource.self;
                    var responseURI = hostname + requestResource.self + "/r
                    // Get result.
                    $.ajax({
                        url: responseURI,
                        // MPS RESTful API (Asynchronous): Specify HTTP GET
                        method: 'GET',
                        dataType: 'json',
                        success: function(response) {
```

```
                                        if (response.hasOwnProperty("lhs")) {
                                            $("#demo").html('<p>' +
                                                response.lhs[0] + '</p>');
                                            //Uncomment the next line if using JSON la
                                            //response.lhs[0].mwdata + '</p>');

                                        } else if (response.hasOwnProperty("error")) {
                                            alert("Error: " + response.error.message);
                                        }
                                        // MPS RESTful API (Asynchronous): Specify HTTP
                                        $.ajax({
                                            url: requestURI,
                                            method: 'DELETE'
                                        });
                                    }
                                });
                            }
                        }
                    });
                }, 200);
            }
        </script>
    </head>

    <body>
        <p><strong>MPS RESTful API and JSON EXAMPLE</strong></p>
        <p> >> mymagic(5)</p>
        <p id="requestURI"></p>
        <p id="state"></p>
        <p id="demo"></p>
        <p> # output from server returned in column-major format </p>
    </body>

</html>
```

JSON Representation of MATLAB Data Types

JSON Representation of MATLAB Data Types

This topic describes the JSON representation of MATLAB data types. JavaScript Object Notation or JSON is a text-based, programming-language independent data interchange format. The JSON standard is defined in RFC 7159 and can represent four primitive types and two structured types. Since JSON is programming language independent, you can represent MATLAB data types in JSON. For more about MATLAB data types, see "Fundamental MATLAB Classes" (MATLAB).

Using the JSON representation of MATLAB data types, you can:

- Represent data or variables in the client code to serve as inputs to the MATLAB function deployed on the server.
- Parse the response from a MATLAB Production Server instance for further manipulation in the client code.

The response from the server contains a JSON `array`, where each element of the array corresponds to an output of the deployed MATLAB function represented as a JSON `object`.

You can represent MATLAB data types in JSON using two formats: *small* and *large*.

- Small format provides a simplified representation of MATLAB data types in JSON. There is a one-to-one mapping between MATLAB data types and their corresponding JSON representation. MATLAB data types that are scalar and of type `double`, `logical`, and `char` can be represented using the small notation. Multidimensional MATLAB arrays of type `double`, `logical`, and `struct` can also be represented using small notation.
- Large format provides a generic representation of MATLAB data types in JSON. The large format uses the JSON `object` notation consisting of property name-value pairs to represent data. You can use large notation for any MATLAB data type that cannot be represented in small notation. The response from the MATLAB Production Server always uses large notation.

A JSON `object` contains the following property name-value pairs:

Property Name	Property Value

`"mwtype"`	JSON `string` representing the type of data. The property value is specified within `""`.
	
	`"double" | "single"`
	`"int8" | "uint8" | "int16" | "uint16"`
	`"int32" | "uint32" | "int64" | "uint64"`
	`"logical" | "char" | "struct" | "cell"`

`"mwsize"`	A JSON `array` representing the dimensions of the data. Specify the property value by enclosing the dimensions as a comma-separated list within `[]` .
`"mwdata"`	JSON `array` representing the actual data. The property value is specified by enclosing the data as a comma-separated list within `[]` .
`"mwcomplex"` *(when representing complex numbers.)*	Set to JSON `true` .

JSON Representation of MATLAB Data Types

In this section...
“Numeric Types: double, single” on page 2-4
“Numeric Types: NaN, Inf, -Inf” on page 2-4
“Numeric Types: Integers” on page 2-6
“Numeric Types: Complex Numbers” on page 2-6
“Characters” on page 2-7
“Logical” on page 2-8
“Cell Arrays” on page 2-8
“Structures” on page 2-9
“Empty Arrays: []” on page 2-10
“Multidimensional Arrays” on page 2-10

Numeric Types: double, single

MATLAB Data Type	JSON Small Notation	JSON Large Notation
`double`	`number`	`{ "mwtype": "double", "mwsize": [1,1], "mwdata": [number] }`
`single`	No small representation.	`{ "mwtype": "single", "mwsize": [1,1], "mwdata": [number] }`
Example:		
`double(12.905)`	`12.905`	`{ "mwtype": "double", "mwsize": [1,1], "mwdata": [12.905] }`
`single(20.15)`	No small representation.	`{ "mwtype": "single", "mwsize": [1,1], "mwdata": [20.15] }`
`42`	`42`	`{ "mwtype": "double", "mwsize": [1,1], "mwdata": [42] }`

Numeric Types: NaN, Inf, -Inf

- `NaN`,`Inf`,`-Inf` are numeric types whose underlying MATLAB class can be either `double` or `single` only. `NaN`, `Inf`,`-Inf` cannot be represented as an integer type in MATLAB.

MATLAB Data Type	JSON Small Notation	JSON Large Notation

`NaN`	`{"mwdata":"NaN"}`	`{`
		`  "mwtype": "double",`
		`  "mwsize": [1,1],`
		`  "mwdata": ["NaN"]`
		`}`
		
		Or
		
		`{`
		`  "mwtype": "double",`
		`  "mwsize": [1,1],`
		`  "mwdata": [{"mwdata":"NaN"}]`
		`}`

`Inf`	`{"mwdata":"Inf"}`	`{`
		`  "mwtype": "double",`
		`  "mwsize": [1,1],`
		`  "mwdata": ["Inf"]`
		`}`
		
		Or
		
		`{`
		`  "mwtype": "double",`
		`  "mwsize": [1,1],`
		`  "mwdata": [{"mwdata":"Inf"}]`
		`}`

`-Inf`	`{"mwdata":"-Inf"}`	`{`
		`  "mwtype": "double",`
		`  "mwsize": [1,1],`
		`  "mwdata": ["-Inf"]`
		`}`
		
		Or
		
		`{`
		`  "mwtype": "double",`
		`  "mwsize": [1,1],`
		`  "mwdata": [{"mwdata":"-Inf"}]`
		`}`

MATLAB Data Type	JSON Small Notation	JSON Large Notation

`[]` empty `double`	`[]`	`{`
		`  "mwtype": "double",`
		`  "mwsize": [0,0],`
		`  "mwdata": [ ]`
		`}`

Numeric Types: Integers

- Integer types from MATLAB cannot be represented using JSON small notation.

MATLAB Data Type	JSON Large Notation

`int8, uint8, int16, uint16`
 	
`int32, uint32, int64, uint64`
	`{`
	`  "mwtype": "int8" | "uint8"  | "int16" | "uint`
	`  "int32" | "uint32" | "int64" | "uint`
	`  "mwsize": [1,1],`
	`  "mwdata": [number]`
	`}`

Example:	

`int8(23)`	`{`
	`  "mwtype": "int8",`
	`  "mwsize": [1,1],`
	`  "mwdata": [23]`
	`}`

`uint8(27)`	`{`
	`  "mwtype": "uint8",`
	`  "mwsize": [1,1],`
	`  "mwdata": [27]}`

Numeric Types: Complex Numbers

- Complex numbers from MATLAB cannot be represented using JSON small notation.
- When representing complex numbers from MATLAB in JSON:
  - A property named `mwcomplex` is added to the JSON `object`, and it's property value is set to `true`.
  - The property values for the `mwdata` property contain the real and imaginary parts represented side-by-side.

MATLAB Data Type	JSON Large Notation

`a + bi`	`{`
	`  "mwtype": "double",`
	`  "mwsize": [1,1],`
	`  "mwcomplex": true,`
	`  "mwdata": [a b]`
	`}`

Example:	

`3 + 4i`	`{`
	`  "mwtype": "double",`
	`  "mwsize": [1,1],`
	`  "mwcomplex": true,`
	`  "mwdata": [3,4]`
	`}`

Characters

MATLAB Data Type	JSON Small Notation	JSON Large Notation

`char`	`string`	`{`
		`  "mwtype": "char",`
		`  "mwsize": [1,1],`
		`  "mwdata": [string]`
		`}`

Example:		

`'a'`	`"a"`	`{`
		`  "mwtype": "char",`
		`  "mwsize": [1,1],`
		`  "mwdata": ["a"]`
		`}`

`'hey, jude'`	`"hey, jude"`	`{`
		`  "mwtype": "char",`
		`  "mwsize": [1,9],`
		`  "mwdata": ["hey, jude"]`
		`}`

Logical

MATLAB Data Type	JSON Small Notation	JSON Large Notation

`logical`	`true | false`	`{`
		`  "mwtype": "logical",`
		`  "mwsize": [1,1],`
		`  "mwdata": [true | false]`
		`}`

Example:		

`logical(1)` or `true`	`true`	`{`
		`  "mwtype": "logical",`
		`  "mwsize": [1,1],`
		`  "mwdata": [true]`
		`}`

`logical(0)` or `false`	`false`	`{`
		`  "mwtype": "logical",`
		`  "mwsize": [1,1],`
		`  "mwdata": [false]`
		`}`

Cell Arrays

MATLAB Data Type	JSON Large Notation

`cell`	`{`
	`  "mwtype": "cell",`
	`  "mwsize": [<cell dimensions>],`
	`  "mwdata": [<cell data>]`
	`}`

Example:	

MATLAB Data Type	JSON Large Notation
`{'Primes', [10 23 199], {false,true,'maybe'}}`	`{ "mwtype": "cell", "mwsize": [1,3], "mwdata": ["Primes", { "mwtype": "double", "mwsize": [1,3], "mwdata": [10,23,199] }, { "mwtype": "cell", "mwsize": [1,3], "mwdata": [false, true,"maybe"] ] }`

Structures

MATLAB Data Type	JSON Small Notation	JSON Large Notation
`struct`		`{ "mwtype": "struct", "mwsize": [<struct dimensions>], "mwdata": [<struct data>] }`
Example:		
`struct('name', 'John Smith', 'age', 15)`	`{"name" : "John Smith", "age" : 15 }`	`{ "mwtype": "struct", "mwsize": [1,1], "mwdata": [{"mwdata":{"age":[{"mw "name":[{"mwdata":["John Smith"] }`

MATLAB Data Type	JSON Small Notation	JSON Large Notation

`x = struct('Name',{{'Casper','Ghost'}},...`
`'Age',{[14,17,18]},...`
`'Date', {736676});`
	No small representation.	`{`
		`"mwtype":"struct",`
		`"mwsize":[1,1],`
		`"mwdata":{`
		`"Name":`
		`[{"mwdata": ,{'Casper','Ghost'},`
		`"mwsize":[1,2],`
		`"mwtype":"string"}],`
		`"Age":`
		`[{"mwdata":[14,17,18],`
		`"mwsize":[1,3],`
		`"mwtype":"double"}],`
		`"Date":`
		`[{"mwdata":[736676],`
		`"mwsize":[1,1],`
		`"mwtype":"double"}]}`
		`}`

Empty Arrays: []

- Empty arrays `[]` cannot be of type `struct`.

MATLAB Data Type	JSON Small Notation	JSON Large Notation

`[]`	`[]`	`{`
		`"mwtype": "double" | "single"`
		`"int8" | "uint8" |`
		`"int32" | "uint32" |`
		`"logical" | "char" |`
		`"mwsize": [0,0],`
		`"mwdata": [ ]`
		`}`

Multidimensional Arrays

Most multidimensional arrays from MATLAB cannot be represented using JSON small notation. The exceptions are: `double`, `logical`, and `struct`. Specify all data from multidimensional arrays in column-major order in the `mwdata` property of the JSON `object`. This ordering corresponds to the default memory layout in MATLAB.

Numeric Types: double, single, NaN, Inf, -Inf, Integers

In the JSON representation of multidimensional numeric arrays:

- The `mwtype` property can take any of the following values:

  ```
  "double" | "single" | "int8" | "uint8" | "int16" | "uint16" |
  "int32" | "uint32" | "int64" | "uint64'
  ```

- The `mwsize` property is specified by enclosing the dimensions as a comma-separated list within `[]` .

MATLAB Data Type	JSON Small Notation	JSON Large Notation

`[1,2,3;...`
` 4,5,6]`
	`[[1,2,3],[4,5,6]]`	`{`
		`  "mwtype": "double",`
		`  "mwsize": [2,3],`
		`  "mwdata": [1,4,2,5,3,6]`
		`}`

`[1, NaN, -Inf;...`
` 2, 105,  Inf]`
	`[[1,{"mwdata": "NaN"},{"mwdata": "-Inf"}],[2,105,{"mwdata": "`	`{`
		`  "mwtype": "double",`
		`  "mwsize": [2,3],`
		`  "mwdata": [1, 2, "NaN", 105, "-In`
		`}`

`[ 1 2; 4 5; 7 8 ]`	`[[1, 2], [4, 5], [7, 8]]`	`{`
		`  "mwtype": "double",`
		`  "mwsize": [3,2],`
		`  "mwdata": [1,4,7,2,5,8]`
		`}`

`a(:,:,1) =`
 	
`  1  2`
`  3  4`
`  5  6`
 	
`a(:,:,2) =`
 	
`  7  8`
`  9  10`
`  11  12`
	`[[[1,7],[2,8]],[[3,9],[4,10]],[[5,11],[6,12]]]`	`{`
		`  "mwtype": "double",`
		`  "mwsize": [3,2,2],`
		`  "mwdata": [1,3,5,2,4,6,7,9,11,8,1`
		`}`

Below is an example of reading and writing multidimensional arrays in column-major order in JavaScript. The example uses a JavaScript file `sub2ind.js` to convert subscripts to linear indices.

Code:

sub2ind.js

```
/*
 * Convert subscripts to linear indices
 *
 * Syntax:
 *
 * linearIndex = sub2ind(dimensions, dim1sub, dim2sub, dim3sub, ...)
 *
 * Example:
 * Call below will return the linear index of (0, 1) from a 2x3 array
 *
 * sub2ind([2, 3], 0, 1)
 */
function sub2ind(dims) {
    var indices = Array.prototype.slice.call(arguments, 1);

    if(dims.length !== indices.length) {
        throw new Error("number of indices must match number of dimensions");
    }
    var size = 1;
    var index = 0;
    for(var i = 0; i < dims.length; i++) {
        index += indices[i] * size;
        size *= dims[i];
    }
      return index;
  }
```

writeReadJsonExample.js

```
/*
 * First: Write a 5x5 magic square to JSON
 * This example uses sub2ind.js
 */
function write_json_example() {
    var data =
        [[17, 24,  1,  8, 15],
         [23,  5,  7, 14, 16],
         [ 4,  6, 13, 20, 22],
         [10, 12, 19, 21,  3],
         [11, 18, 25,  2,  9]];
```

```
    var mwsize = [5, 5];
    var mwdata = []
    for(var r = 0; r < mwsize[0]; r++) {
        for(var c = 0; c < mwsize[1]; c++) {
            mwdata[sub2ind(mwsize, r, c)] = data[r][c];
        }
    }

    var json = JSON.stringify({ 'mwtype' : 'int8', 'mwsize' : mwsize, 'mwdata' : mwdata
    return json;
}
/*
 * Second: Read 5x5 magic square
 * This example uses sub2ind.js
*/
function read_json_example() {
    var json = JSON.parse('{"mwtype":"int8","mwsize":[5,5],"mwdata":[17,23,4,10,11,24,5
    var mwdata = json.mwdata;
    var mwsize = json.mwsize;

    var data = [];
    for(var r = 0; r < mwsize[0]; r++) {
        data[r] = [];
        for(var c = 0; c < mwsize[1]; c++) {
            data[r][c] = mwdata[sub2ind(mwsize, r, c)];
        }
    }

    return data;
}
```

Numeric Types: Complex Numbers

MATLAB Data Type	JSON Large Notation

`[1 - 2i;...`
` 3 + 7i]`
	`{`
	`  "mwtype":  "double",`
	`  "mwsize":  [2,1],`
	`  "mwcomplex": true,`
	`  "mwdata":  [1, -2, 3, 7]`
	`}`

Characters

In the JSON representation of multidimensional character arrays:

- The `mwtype` property must have a value of `char` .
- The `mwdata` property must be an array of JSON `strings` .

MATLAB Data Type	JSON Large Notation

`['boston';...`
` '123456']`
	`{`
	`  "mwtype": "char",`
	`  "mwsize": [3,4],`
	`  "mwdata": ["b1o2s3t4o5n6"]`
	`}`

Logical

In the JSON representation of multidimensional logical arrays:

- The `mwtype` property must have a value of `logical` .
- The `mwdata` property must contain only JSON `true|false` values.

MATLAB Data Type	JSON Small Notation	JSON Large Notation

`[true,false;...`
` true,false;...`
` true,false]`
	`[[[true,false],[true,false],[true,false]]]`	`{`
		`  "mwtype": "logical",`
		`  "mwsize": [3,2],`
		`  "mwdata": [true,true,true,false,f`
		`}`

Cell Arrays

In the JSON representation of multidimensional cell arrays:

- The `mwtype` property must have a value of `cell` .
- The `mwdata` property must be a JSON `array` that contains the values of the cells in their JSON representation.

MATLAB Data Type	JSON Large Notation
`{ 'hercule', 18540, [33 1 50];... {'agatha',1920,true}, false, 1950 }`	`{ "mwtype": "cell", "mwsize": [2,3], "mwdata": ["hercule", {"mwtype": "cell", "mwsize": [1,3], "mwdata": ["agatha", 1920, true] }, 18540, false, {"mwtype": "double", "mwsize": [1,3], "mwdata": [33,1,50] },1950 ] }`

Structures

In the JSON representation of multidimensional structure arrays:

- The `mwdata` is a JSON `object` containing property name-value pairs.
- The name in each property name-value pair matches a *field* in the structure array.
- The value in each property name-value pair is a JSON `array` containing values for that field for every element in the structure array. The elements of the JSON `array` must be in column-major order.

MATLAB Data Type	JSON Large Notation
`struct( 'Name',{'Casper','Ghost';... 'Genie' ,'Wolf'},... 'Ages',{14,17;... 20,23} )`	`{ "mwtype": "struct", "mwsize": [2,2], "mwdata": {"Name": ["Casper", "Genie", "Ghost" , "Wolf"], "Ages": [14,20, 17,23] } }`

Troubleshooting RESTful API Errors

Troubleshooting RESTful API Errors

Since communication between the client and MATLAB Production Server is over HTTP, many errors are indicated by an HTTP status code. Errors in the deployed MATLAB function use a different format. See “Structure of MATLAB Error” on page 3-4 for more information. To review API usage, see “RESTful API” on page 1-2.

HTTP Status Codes

400–Bad Request

Message	Description
`Invalid input`	Client request is not formatted correctly.
`Invalid JSON`	Client request does not contain a valid JSON representation.
`nargout missing`	Client request does not specify `nargout` containing output arguments.
`rhs missing`	Client request does not specify `rhs` containing input arguments.
`Invalid rhs`	Input arguments does not follow the JSON representation for MATLAB data types.

403–Forbidden

Message	Description
`The client is not authorized to access the requested component`	Client does not have the correct credentials to make a request.

404–Not Found

Message	Description
`Function not found`	Server could not find the MATLAB function in the deployed CTF archive.
`Component not found`	Was unable to find the CTF archive.
`URI-path not of form '/APPLICATION/FUNCTION'`	URL not in the correct format.

405–Method Not Allowed

Message	Description
`Bad Method`	Method is not allowed.
`Method must be POST`	Method is not allowed.
`Unsupported method`	Method is not allowed.

411–Length Required

Message	Description
`Content-length missing`	Length of the content is missing.

415–Unsupported Media Type

Message	Description
`<VALUE> is not an accepted content type`	Did not set correct content type for JSON.

500–Internal Server Error

Message	Description
`Function return type not supported`	MATLAB function deployed on the server returned a MATLAB data type that MATLAB Production Server does not support.

Resource Query vs Resource States

Resources / Server States	NOT_FOUND	READING	IN_QUEUE	PROCESSING	READY	ERROR	CANCELLED	DELETED / PURGED	UNKNOWN SERVER ERROR
GET $request-uri/result	404 - RequestNotFound	204 - NoContent	204 - NoContent	204 - NoContent	200 - OK	200 - OK	410 - RequestAlreadyCancelled	410 - RequestAlreadyDeleted	500 - InternalServerError
POST $request-uri/cancel	404 - RequestNotFound	204 - NoContent	204 - NoContent	204 - NoContent	410 - RequestAlreadyCompleted	410 - RequestAlreadyCompleted	410 - RequestAlreadyCancelled	410 - RequestAlreadyDeleted	500 - InternalServerError
DELETE $request-uri	404 - RequestNotFound	409 - RequestNotCompleted	409 - RequestNotCompleted	409 - RequestNotCompleted	204 - NoContent	204 - NoContent	204 - NoContent	410 - RequestAlreadyDeleted	500 - InternalServerError

Structure of MATLAB Error

In order to resolve a MATLAB error, you will need to troubleshoot the MATLAB function deployed on the server.

```
{"error": {
   "type": "matlaberror",
   "id": error_id,
   "message": error_message,
   "stack": [
     {"file": file_name1,
      "name": function_name1,
      "line": file_line_number1},
     {"file": file_name2,
      "name": function_name2,
      "line": file_line_number2},
     ...]}}
```

Access-Control-Allow-Origin

Client programmers using JavaScript need to verify whether Cross-Origin Resource Sharing (CORS) is enabled on a MATLAB Production Server instance if their clients programs will be making requests from different domains. If CORS is not enabled, you may get the following error message:

```
Response to preflight request doesn't pass access control check: No 'Access-Control-Al
```

For information on how to enable CORS, see `cors-allowed-origins`.

Examples: RESTful API and JSON

Example: Web-Based Bond Pricing Tool Using JavaScript

This example shows how to create a web application that calculates the price of a bond from a simple formula. It uses the MATLAB Production Server RESTful API on page 1-2 and "JSON Representation of MATLAB Data Types" on page 2-2 to depict an end-to-end workflow of using MATLAB Production Server. You run this example by entering the following known values into a web interface:

- Face value (or value of bond at maturity) — `M`
- Coupon payment — `C`
- Number of payments — `N`
- Interest rate — `i`

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

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

You can use the sliders in the web application to price different bonds.

In this section...

"Step 1: Write MATLAB Code" on page 4-2
"Step 2: Create a Deployable Archive with the Production Server Compiler App" on page 4-3
"Step 3: Place the Deployable Archive on a Server" on page 4-3
"Step 4: Enable Cross-Origin Resource Sharing (CORS) on the Server" on page 4-3
"Step 5: Write JavaScript Code using the RESTful API and JSON" on page 4-4
"Step 6: Embed JavaScript within HTML Code" on page 4-5
"Step 7: Run Example" on page 4-7

Step 1: Write MATLAB Code

Write the following code in MATLAB to price bonds. Save the code using the filename `pricecalc.m`.

```
function price = pricecalc(face_value, coupon_payment,...
                           interest_rate, num_payments)
    M = face_value;
```

```
C = coupon_payment;
N = num_payments;
i = interest_rate;

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

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

To create the deployable archive for this example:

1. On the **Apps** tab, select the Production Server Compiler App.
2. In the **Application Type** list, select **Deployable Archive**.
3. In the **Exported Functions** field, add `pricecalc.m`.
4. Under **Archive information**, change `pricecalc` to `BondTools`.
5. Click **Package**.

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

Step 3: Place the Deployable Archive on a Server

1. Download the MATLAB Runtime, if needed, at http://www.mathworks.com/products/compiler/mcr. See “Download and Install the MATLAB Runtime” for more information.
2. Create a server using `mps-new`. See “Create a Server” for more information. If you haven't already setup your server environment, see `mps-setup` for more information.
3. If you have not already done so, specify the location of the MATLAB Runtime to the server by editing the server configuration file, `main_config` and specifying a path for `--mcr-root`. See “Edit the Configuration File” for details.
4. Start the server using `mps-start`, and verify it is running with `mps-status`.
5. Copy the `BondTools.ctf` file to the `auto_deploy` folder on the server for hosting.

Step 4: Enable Cross-Origin Resource Sharing (CORS) on the Server

Enable Cross-Origin Resource Sharing (CORS) by editing the server configuration file, `main_config` and specifying the list of domains origins from which requests can be

made to the server. For example, setting the `cors-allowed-origins` option to `--cors-allowed-origins *` allows requests from any domain to access the server. See `cors-allowed-origins` and "Edit the Configuration File" for details.

Step 5: Write JavaScript Code using the RESTful API and JSON

Using the RESTful API on page 1-2 and JSON Representation of MATLAB Data Types on page 2-2 as a guide, write the following JavaScript code. Save this code as a JavaScript file named `calculatePrice.js`.

Code:

calculatePrice.js

```
//calculatePrice.js : JavaScript code to calculate the price of a bond.
        function calculatePrice()
        {
            var cp = parseFloat(document.getElementById('coupon_payment_value').value);
            var np = parseFloat(document.getElementById('num_payments_value').value);
            var ir = parseFloat(document.getElementById('interest_rate_value').value);
            var vm = parseFloat(document.getElementById('facevalue_value').value);

            // A new XMLHttpRequest object
            var request = new XMLHttpRequest();
            //Use MPS RESTful API to specify URL
            var url = "http://localhost:9910/BondTools/pricecalc";

            //Use MPS RESTful API to specify params using JSON
            var params = { "nargout":1,
                           "rhs": [vm, cp, ir, np] };

            document.getElementById("request").innerHTML = "URL: " + url + "<br>"
                    + "Method: POST <br>" + "Data:" + JSON.stringify(params);

            request.open("POST", url);

            //Use MPS RESTful API to set Content-Type
            request.setRequestHeader("Content-Type", "application/json");

            request.onload = function()
            {   //Use MPS RESTful API to check HTTP Status
                if (request.status == 200)
                {
```

```
                // Deserialization: Converting text back into JSON object
                // Response from server is deserialized
                var result = JSON.parse(request.responseText);

                //Use MPS RESTful API to retrieve response in "lhs"
                if('lhs' in result)
                {  document.getElementById("error").innerHTML = "" ;
                   document.getElementById("price_of_bond_value").innerHTML = "Bond
                else { document.getElementById("error").innerHTML = "Error: " + res
            }
            else { document.getElementById("error").innerHTML = "Error:" + request.
            document.getElementById("response").innerHTML = "Status: " + request.st
                    + "Status message: " + request.statusText + "<br>" +
                    "Response text: " + request.responseText;
        }
        //Serialization: Converting JSON object to text prior to sending request
        request.send(JSON.stringify(params));
    }

    //Get value from slider element of "document" using its ID and update the value
    //The "document" interface represent any web page loaded in the browser and
    //serves as an entry point into the web page's content.
    function printValue(sliderID, valueID) {
        var x = document.getElementById(valueID);
        var y = document.getElementById(sliderID);
        x.value = y.value;
    }
    //Execute JavaScript and calculate price of bond when slider is moved
    function updatePrice(sliderID, valueID) {
        printValue(sliderID, valueID);
        calculatePrice();
    }
```

Step 6: Embed JavaScript within HTML Code

Embed the JavaScript from the previous step within the following HTML code by using the following syntax:

```
<script src="calculatePrice.js" type="text/javascript"></script>
```

Save this code as an HTML file named `bptool.html`.

Code:

bptool.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head lang="en">
    <meta charset="UTF-8">
    <title>Bond Pricing Tool</title>
</head>
<body>
<!-- Embed the JavaScript code here by referencing calculatePrice.js -->
<!-- <script src="calculatePrice.js" type="text/javascript"></script> -->
    <script>
        //Helper Code: Execute JavaScript immediately after the page has been loaded
        window.onload = function() {
            printValue('coupon_payment_slider', 'coupon_payment_value');
            printValue('num_payments_slider', 'num_payments_value');
            printValue('interest_rate_slider', 'interest_rate_value');
            printValue('facevalue_slider', 'facevalue_value');
            calculatePrice();
        }
    </script>
    <h1><a>Bond Pricing Tool</a></h1>
    <h2></h2>
    This example shows an application that calculates a bond price from a simple formul
    You run this example by entering the following known values into a simple graphical
      <ul>
        <li>Face Value (or value of bond at maturity) - M</li>
        <li>Coupon payment - C</li>
        <li>Number of payments - N</li>
        <li>Interest rate - i</li>
      </ul>
      The application calculates price (P) based on the following equation:<p>
      P = C * ( (1 - (1 + i)^-N) / i ) + M * (1 + i)^-N<p>
      <hr>
      <h3>M: Face Value </h3>
      <input id="facevalue_value" type="number" maxlength="4" oninput="updatePrice('fac
      <input type="range" id="facevalue_slider" value="0" min="0" max="10000" onchange=

      <h3>C: Coupon Payment </h3>
      <input id="coupon_payment_value" type="number" maxlength="4" oninput="updatePrice
      <input type="range" id="coupon_payment_slider" value="0" min="0" max="1000" oncha

      <h3>N: Number of payments  </h3>
      <input id="num_payments_value" type="number" maxlength="4" oninput="updatePrice(
```

```
      <input type="range" id="num_payments_slider" value="0" min="0" max="1000" onchang

      <h3>i: Interest rate </h3>
      <input id="interest_rate_value" type="number" maxlength="4" step="0.01" oninput="
      <input type="range" id="interest_rate_slider" value="0" min="0" max="1" step="0.0

    <h2>BOND PRICE</h2>
    <p id="price_of_bond_value" style="font-weight: bold">
    <p id="error" style="color:red">

    <hr>
    <h3>Request to MPS Server</h3>
    <p id="request">

    <h3>Response from MPS Server</h3>
    <p id="response">
    <hr>
</body>
</html>
```

Step 7: Run Example

Assuming, the server with the deployed MATLAB function is up and running, open the HTML file `bptool.html` in a web browser. The default bond price is `NaN` because no values have been entered as yet. Try the following values to price a bond:

- Face Value = $1000
- Coupon Payment = $100
- Number of payments = 5
- Interest rate = 0.08 *(Corresponds to 8%)*

The resulting bond price is $1079.85

You can use the sliders in the tool price different bonds. Varying the interest rate results in the most dramatic change in the price of the bond.

Bond Pricing Tool

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:

- Face Value (or value of bond at maturity) — M
- Coupon payment — C
- Number of payments — N
- Interest rate — i

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

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

M: Face Value

1000

C: Coupon Payment

100

N: Number of payments

5

i: Interest rate

0.08

BOND PRICE

$: 1079.8542007415617

Request to MPS Server

URL: http://localhost:9910/BondTools/pricecalc
Method: POST
Data:{"nargout":1,"rhs":[1000,100,0.08,5]}

Response from MPS Server

5

RESTful API Reference