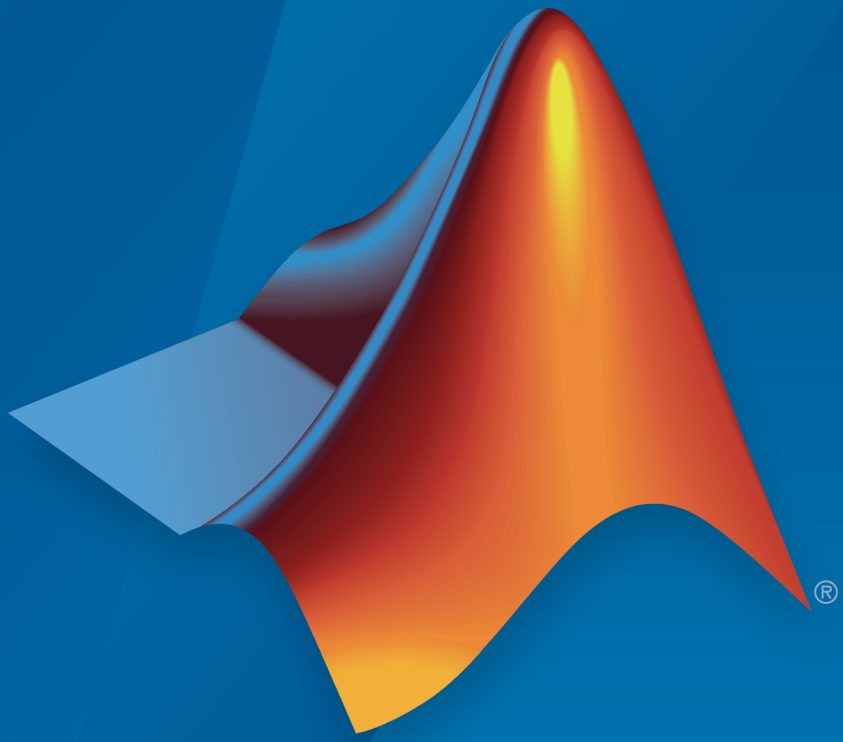
MATLAB® Production Server™

RESTful API and JSON



MATLAB®

R2016b

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)

Contents

1 Client Programming

2 JSON Representation of MATLAB Data Types

3 Troubleshooting RESTful API Errors

4 Examples: RESTful API and JSON

1

Client Programming

RESTful API

In this section...

This topic describes the MATLAB® Production Server™ RESTful API. The API operates in both synchronous and asynchronous modes:

- In the synchronous mode, execution provides an immediate response to a query. In this situation, the client will wait until the server sends back the response message.
- In the asynchronous mode, execution may require a significant amount of time to process a request. However, the client can continue with other processing in the meantime rather than wait for the response.

The RESTful API consists of two phases: a request phase where data is relayed from the client to the server and a response phase where data is returned from the server to the client.

Synchronous

Request

Element	Option	Description
URL	`http://hostName:portNumber/CTF_archiveName/MATLAB_functionName`	URL to the deployed MATLAB function.
HTTP method	`POST`	Post MATLAB execution request
Content-Type	`application/json`	Media type of the document.
Request Payload	Single Input Argument: {	Input arguments to the deployed MATLAB function. The option varies depending on whether a

Element	Option	Description

	"nargout": num_of_requested_o
	"rhs": arg1,
	"outputFormat" : { "mode" : "
	"nanInfFor
	}
	Multiple Input Arguments:
	
	{
	"nargout": num_of_requested_o
	"rhs": [arg1, arg2, ...],
	"outputFormat" : { "mode" : "
	"nanInfFor
	}
		single input argument or multiple arguments are being passed.
		•
		• nargout specifies the number of outputs that the client is requesting from the deployed MATLAB function. MATLAB functions, depending on their intended purpose, can be coded to return multiple outputs. A subset of the potential outputs which can be specified using nargout.
		• rhs specifies the input to the deployed MATLAB function. Multiple inputs are specified as an array of comma-separated values.
		• outputFormat (optional) Describes the format used for representation of MATLAB data in the result of MATLAB execution request. The possible values of mode are smalland large. If this option is not used, the MATLAB data in the result will be in the large notation. The nanInfFormat option decides how the MATLAB NaN and Inf values are represented.

Response

Element	Option	Description
Response body	{ "lhs": [output1, output2, ..	Representation of the output from the server.

Element	Option	Description
	}	• `lhs` is a JSON `array` contained in the response from the server. Each element of the JSON `array` corresponds to an output of the deployed MATLAB function represented using JSON large notation. For more information on JSON large notation see "JSON Representation of MATLAB Data Types" on page 2-2. • Client code can index into the `lhs` of the JSON `array` and further manipulate a particular output. • Client code needs to check if `lhs` is present in the response from the server prior to parsing it.
HTTP status code	`200`	HTTP status code `200` indicates that the server has accepted and processed the request. Client code needs to check against this status code and whether `lhs` is present in the response from the server prior to parsing it. If you encounter a status code other than `200`, see "Troubleshooting RESTful API Errors" on page 3-2.

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® and the other using Python®. 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

The JavaScript implementation of the RESTful API involves including the script within the `<script> </script>` tags of an HTML page. When this HTML page is opens 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.

```
<!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 s
                        document.getElementById("demo").innerHTML = '<p>' + result.lhs[
                    }
                        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

```
#!/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"]))
```

For an example illustrating the complete workflow of deploying a MATLAB function on MATLAB Production Server and invoking it using the RESTful API and JSON see “Example: Web-based Bond Pricing Tool Using JavaScript” on page 4-2

Asynchronous

Request 1: Initiate a MATLAB Execution Request

Element	Option	Description
URL	`http://hostName:portNumber/CTF_archiveName/MATLAB_functionName`	URL to the deployed MATLAB function
Query parameters	`mode=async` `client=<client-id>`	Specify the mode of communication between the client and server. If you do not specify the `mode=async` parameter, the request will default to a synchronous request. • `mode` represents the mode of communication (required). • `client` represents the client ID specified in the request (optional).
HTTP method	`POST`	Post MATLAB execution request
Content-Type	`application/json`	Media type of the document

Request Payload	Single Input Argument:
	`{`
	` "nargout": num_of_requested_o`
	` "rhs": arg1,`
	` "outputFormat" : { "mode" : "`
	`"nanInfFor`
	`}`
	Multiple Input Arguments:
	`{`
	` "nargout": num_of_requested_o`
	` "rhs": [arg1, arg2, ...],`
	` "outputFormat" : { "mode" : "`
	`"nanInfFor`
	`}`
		Input arguments to the deployed MATLAB function. The option varies depending on whether a single input argument or multiple arguments are being passed.
		• `nargout` specifies the number of outputs that the client is requesting from the deployed MATLAB function. MATLAB functions, depending on their intended purpose, can be coded to return multiple outputs. A subset of the potential outputs which can be specified using `nargout`.
		• `rhs` specifies the input to the deployed MATLAB function. Multiple inputs are specified as

Element	Option	Description
		an array of comma-separated values. • `outputFormat` (optional) Describes the format used for representation of MATLAB data in the result of MATLAB execution request. The possible values of `mode` are `small` and `large`. If this option is not used, the MATLAB data in the result will be in the `large` notation. The `nanInfFormat` option decides how the MATLAB `NaN` and `Inf` values are represented.

Response 1: A Representation of MATLAB Execution Request

Element	Option	Description
HTTP status code	`201`	HTTP status code `201` indicates that the server has accepted the request. Client code needs to check against this status code. If you encounter a status code other than `201`, see "Troubleshooting RESTful API Errors" on page 3-2.

Response body	`{`
	`  "id": <id-string>,`
	`  "self": <request-uri-string>`
	`  "up": <request-collection-u`
	`  "lastModifiedSeq": <server-s`
	`  "state": <request-state-str`
	`  "client": <client-id-string>`
	`}`
		• `id` represents a unique request ID.
		• `self` represents request resource uri.
		• `up` represents a collection of requests.
		• `lastModifiedSeq` indicates when this request represented by `self` was last modified.

Element	Option	Description
		• `state` indicates the state of a request. The following is a list of request states: `READING`, `IN_QUEUE`, `PROCESSING`, `READY`, `ERROR`, and `CANCELLED`. • `client` represents the client ID specified in the request. This will be an empty string if no `client` query parameter was provided as part of the request.

Request 2: Get the Representation of MATLAB Execution Request

Element	Option	Description
URL	`http://hostName:portNumber/<request-uri-string>`	URL to the individual request
HTTP method	`GET`	Get information of one request. You can query for more than one requests using Request 3.
Content-Type	`application/json`	Media type of the document

Response 2: A Representation of MATLAB Execution Request

Element	Option	Description
HTTP status code	`200`	Client code needs to check against this status code. If you encounter a status code other than `200`, see "Troubleshooting RESTful API Errors" on page 3-2.

Response body	`{`
	`  "id": <id-string>,`
	`  "self": <request-uri-string>`
	`  "up": <request-collection-u`
	`  "lastModifiedSeq": <server-s`
	`  "state": <request-state-str`
	`  "client": <client-id-string>`
	`}`
		• `id` represents a unique request ID.
		• `self` represents request resource uri.
		• `up` represents a collection of requests.

Element	Option	Description
		• `lastModifiedSeq` indicates when this request represented by `self` was last modified. • `state` indicates the state of a request. The following is a list of request states: `READING`, `IN_QUEUE`, `PROCESSING`, `READY`, `ERROR`, and `CANCELLED`. • `client` represents the client ID specified in the request.

Request 3: Query for a Collection of MATLAB Execution Requests

Element	Option	Description
URL	`http://hostName:portNumber/<request-collection-uri-string`	URL to the collection of requests

Query parameters	`since=<server-state-number>`
	`clients=<client-id>,<client-id`
	`ids=<id>,<id>, ...`
		• `since` represents the starting server state number of updates yet to be seen (required).
		• `clients` represents one or more client ID strings. (required if `ids` is not specified).
		• `ids` represents one or more request ID strings. (required if `clients` is not specified).

HTTP method	`GET`	Get information of one or more requests
Content-Type	`application/json`	Media type of the document

Response 3: A Collection of MATLAB Execution Requests that Matched the Query Parameter

Element	Option	Description
HTTP status code	`200`	Client code needs to check against this status code. If you encounter

Element	Option	Description
		a status code other than `200`, see "Troubleshooting RESTful API Errors" on page 3-2.

Response body	`{`
	`  "createdSeq': <server-state`
	`  "data": [`
	`  {`
	`  "id": <id-string>,`
	`  "self": <request-uri-string`
	`  "up": <request-collection-u`
	`  "lastModifiedSeq": <server-`
	`  "state": <request-state-str`
	`  "client": <client-id-string`
	`  },`
	`  {`
	`  "id": <id-string>,`
	`  "self": <request-uri-string`
	`  "up": <request-collection-u`
	`  "lastModifiedSeq": <server-`
	`  "state": <request-state-str`
	`  "client": <client-id-string`
	`  },...`
	`  ]`
	`  }`
		• `createdSeq` corresponds to server state number. The requests included in the `data` collection are the requests that have gone through some state change between `since` and `createdSeq`.
		• `data` represents a collection of MATLAB execution requests that matched the query.
		• `id` represents a unique request ID.
		• `self` represents request resource uri.
		• `up` represents a collection of requests.
		• `lastModifiedSeq` indicates when this request represented by `self` was last modified.
		• `state` indicates the state of a request. The following is a list of request states: `READING`, `IN_QUEUE`, `PROCESSING`, `READY`, `ERROR`, and `CANCELLED`.
		• `client` represents the client ID specified in the request.

Request 4: Get State Information of MATLAB Execution Request

Element	Option	Description

URL	`http://`
	`hostName:portNumber/`
	`<request-uri-string>/info`
		URL to the individual request information

Element	Option	Description
HTTP method	`GET`	Get information of one request
Content-Type	`application/json`	Media type of the document

Response 4: A Representation of MATLAB Execution Request's State Information

Element	Option	Description
HTTP status code	`200`	Client code needs to check against this status code. If you encounter a status code other than `200`, see "Troubleshooting RESTful API Errors" on page 3-2.

Response body	`{`
	`"request": <request-uri-str`
	`"lastModifiedSeq": <server-`
	`"state": <request-state-str`
	`}`
		• `request` represents request resource uri.
		• `lastModifiedSeq` indicates when this request represented by `request` was last modified.
		• `state` indicates the state of a request. The following is a list of request states: `READING`, `IN_QUEUE`, `PROCESSING`, `READY`, `ERROR`, and `CANCELLED`.

Request 5: Get the Result of MATLAB Execution Request

Element	Option	Description

URL	`http://`
	`hostName:portNumber/`
	`<request-uri-string>/result`
		URL to the individual MATLAB execution request result

HTTP method	`GET`	Get result of a MATLAB execution request
Content-Type	`application/json`	Media type of the document

Response 5: Result Output from MATLAB Execution Request

Element	Option	Description
HTTP status code	`200`	Client code needs to check against this status code. If you encounter

Element	Option	Description
		a status code other than `200`, see "Troubleshooting RESTful API Errors" on page 3-2.

Response body	`{`
	`  "lhs": [output1, output2, .`
	`}`
		Representation of the output from the server.
		• `lhs` is a JSON `array` contained in the response from the server. Each element of the JSON `array` corresponds to an output of the deployed MATLAB function represented using JSON large notation. For more information on JSON large notation see "JSON Representation of MATLAB Data Types" on page 2-2.
		• Client code needs to check if `lhs` is present in the response from the server prior to parsing it.

	`{`
	` "error":`
	`  { "type": "matlaberror",`
	`  "id": <matlab-error-id>,`
	`  "message": <matlab-error-m`
	`  "stack":`
	`  [`
	`  { "file": <matlab-file-st`
	`  "name": <matlab-functio`
	`  "line": <matlab-line-nu`
	`  },`
	`... ]`
	`  }`
	`}`
		MATLAB Error with MATLAB stack

	`{`
	`  "error":`
	`  { "type": "httperror",`
	`  "code":<http-error-code-nu`
		HTTP Error with HTTP code and message

Element	Option	Description

	`"message": <http-msg-stri`
	`}`
	`}`	

Request 6: Cancel the MATLAB Execution Request

Element	Option	Description
URL	`http://hostName:portNumber/<request-uri-string>/cancel`	URL to the MATLAB execution request cancellation
HTTP method	`POST`	Post the cancellation
Content-Type	`application/json`	Media type of the document

Response 6: Response to Indicate that the Cancellation Request has been Accepted

Element	Option	Description
HTTP status code	`204`	Client code needs to check against this status code. If you encounter a status code other than `204`, see "Troubleshooting RESTful API Errors" on page 3-2.
Response body		There is no response body.

Request 7: Delete the MATLAB Execution Request

Element	Option	Description
URL	`http://hostName:portNumber/<request-uri-string>`	URL to the MATLAB execution request
HTTP method	`DELETE`	Delete the request from server. Once deleted the request is not available for further querying.
Content-Type	`application/json`	Media type of the document

Response 7: Response to Indicate that the Deletion Request has been Accepted

Element	Option	Description
HTTP status code	204	Client code needs to check against this status code. If you encounter a status code other than 204, see “Troubleshooting RESTful API Errors” on page 3-2.
Response body		There is no response body.

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”.

```
<!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=" + mod
            // 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/
                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>' + requestResou


                            if (requestResource.state != "READY" && requestResource.st
                                //Keep polling if the request is not done yet.
                                pollUsingUp(requestResource);
                            } else {
                                var requestURI = hostname + requestResource.self;
                                var responseURI = hostname + requestResource.self + "/
                                // Get result.
```

```
                                $.ajax({
                                    url: responseURI,
                                    // MPS RESTful API (Asynchronous): Specify HTTP GE
                                    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 HTT
                                        $.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>
```

2

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".

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 `""`.

Property Name	Property Value
	`"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`

MATLAB Data Type	JSON Small Notation	JSON Large Notation
`double, single`	`number`	`{`

MATLAB Data Type	JSON Small Notation	JSON Large Notation
		`{ "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"] }`

MATLAB Data Type	JSON Small Notation	JSON Large Notation

		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"}]
		}

`[]` 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],`

MATLAB Data Type	JSON Large Notation

	`"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]`
		`}`

MATLAB Data Type	JSON Small Notation	JSON Large Notation

`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:	

`{'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:		

MATLAB Data Type	JSON Small Notation	JSON Large Notation

`struct('name', 'John Smit`	`{ "name" : "John Smith",`	`{`
		`  "mwtype": "struct",`
		`  "mwsize": [1,1],`
		`  "mwdata": [{"mwdata":{"age":[{"mv`
		`"name":[{"mwdata":["John Smith"],"mv`
		`}`

`x = struct('Name',{'Caspe`
`'Ages',{14,17}`
	No small representation.	`{`
		`  "mwtype": "struct",`
		`  "mwsize": [1,2],`
		`  "mwdata": {"Name": ["Casper", "Gh`
		`  "Ages": [14,17]`
		`  }`
		`}`

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"},{"m`	`{ 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,`	`{`
		`  "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.

```
/*
 * 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;
  }

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;
}

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,fals`	`{`
		`  "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`
`  {'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]`
	`}`
	`}`

3

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

Resource Server States	NOT_FO	READING	IN_QUEU	PROCESS	READY	ERROR	CANCELL	DELETED PURGED	UNKNOWN SERVER ERROR
GET $reques uri/ result	404 - Requesi	204 - NoConte	204 - NoConte	204 - NoConte	200 - OK	200 - OK	410 - Requesi	410 - Requesi	500 - InternalDeleted
POST $reques uri/ cancel	404 - Requesi	204 - NoConte	204 - NoConte	204 - NoConte	410 - Requesi	410 - Requesi	410 - Requesi	410 - Requesi	500 - InternalDeleted

Resource Server States	NOT_FOUND	READING	IN_QUEUE	PROCESSING	READY	ERROR	CANCELLED	DELETED PURGED	UNKNOWN SERVER ERROR
DELETE $request uri	404 - Request	409 - Request	409 - Request	409 - Request	204 - NoConte	204 - NoConte	204 - NoConte	410 - Request	500 - InternalServerEr

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},
     ...]}}
```

4

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 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

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 and JSON Representation of MATLAB Data Types as a guide, write the following JavaScript code. Save this code as a JavaScript file named `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 = "Bon
                    else { document.getElementById("error").innerHTML = "Error: " + re
```

```
                }
                else { document.getElementById("error").innerHTML = "Error:" + request
                document.getElementById("response").innerHTML = "Status: " + request.s
                        + "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 valu
        //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`.

```
<!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 formu
    You run this example by entering the following known values into a simple graphica
      <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('fa
      <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="updatePric
      <input type="range" id="coupon_payment_slider" value="0" min="0" max="1000" onch

      <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" onchan

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