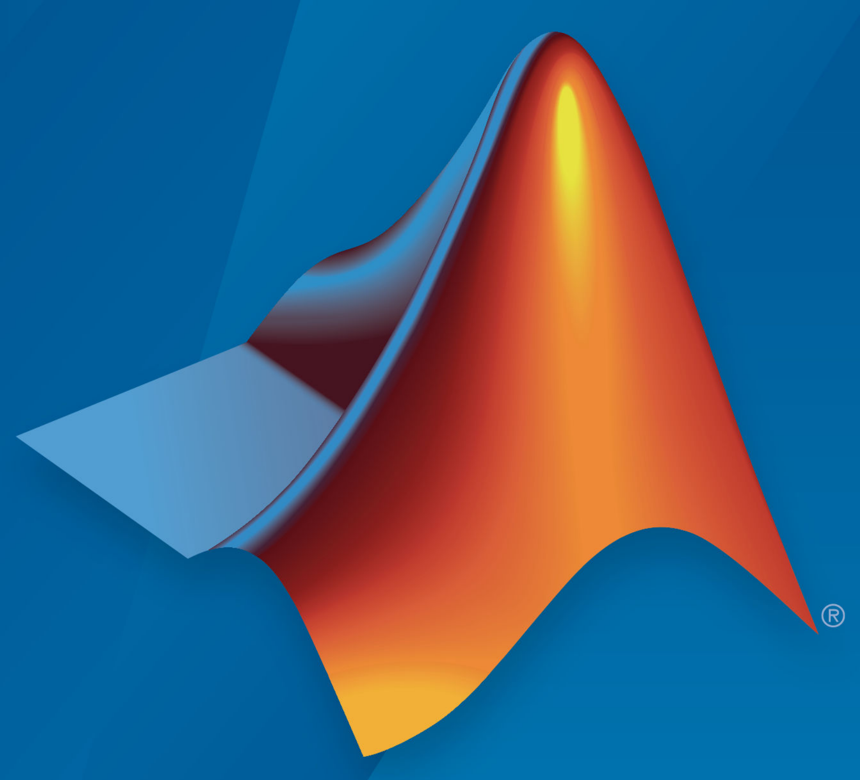
MATLAB® Compiler SDK™

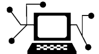
RESTful API and JSON



MATLAB®

R2018a

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® Compiler SDK ™ MATLAB® Production Server ™ RESTful API and JSON

© COPYRIGHT 2016-2018 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

March 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 R2017a)
September 2017	Online only	Revised for Version 6.4 (Release R2017b)
March 2018	Online only	Revised for Version 6.5 (Release R2018a)

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 **1-9**
Discovery Service **1-11**

MATLAB Function Signatures in JSON **1-17**
Function Objects **1-19**
Signature Objects **1-19**
Argument Objects **1-19**
Typedef Object **1-21**

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 **2-5**
Numeric Types: Integers **2-6**
Numeric Types: Complex Numbers **2-7**
Characters **2-8**
Logical **2-8**
Cell Arrays **2-9**
Structures **2-10**
Empty Arrays: [] **2-11**
Multidimensional Arrays **2-11**

3 Troubleshooting RESTful API Errors

Troubleshooting RESTful API Errors **3-2**
Structure of MATLAB Error **3-4**
Structure of HTTP Error **3-5**
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 **4-4**
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

6 Functions — Alphabetical List

1

Client Programming

RESTful API

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.

API

Type	Purpose
"Synchronous Execution" on page 1-2	Make a synchronous requests to the server
"Asynchronous Execution" on page 1-6	Make a asynchronous requests to the server
"Discovery Service" on page 1-11	Discover MATLAB functions deployed on the server

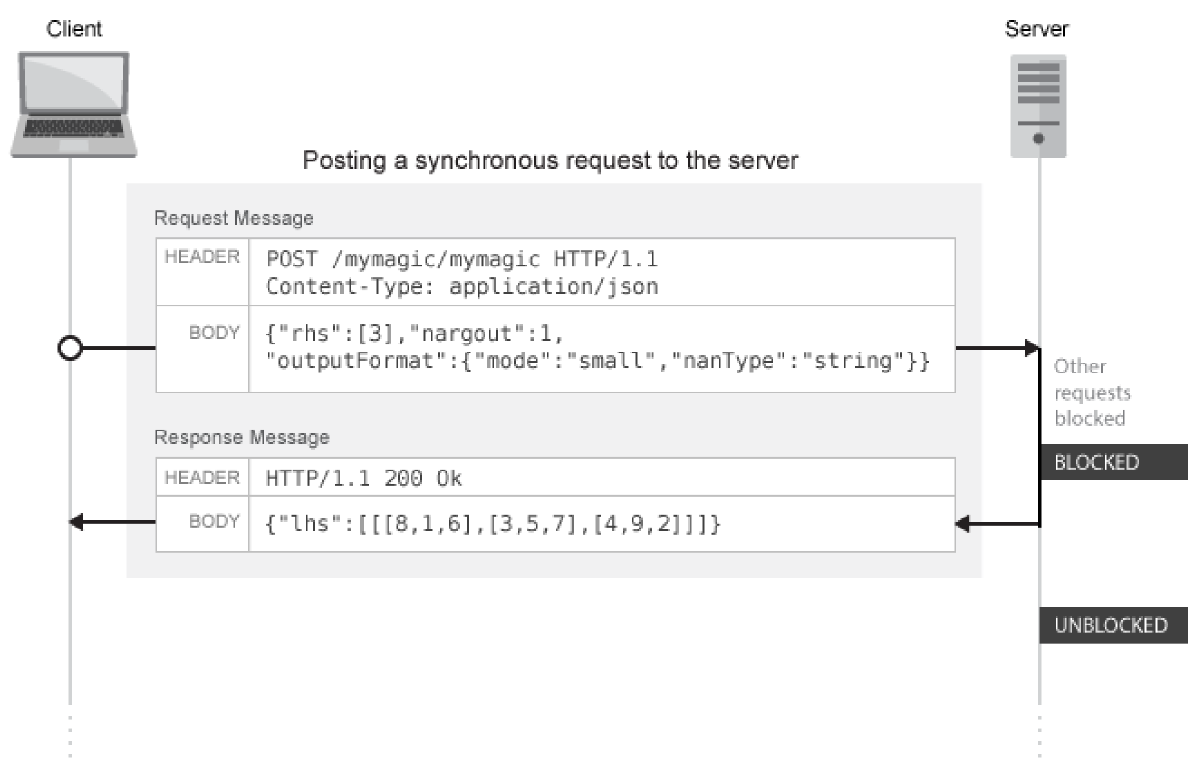
Synchronous Execution

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” (MATLAB Production Server). For more information on setting up a server, see “Create a Server” (MATLAB Production 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.

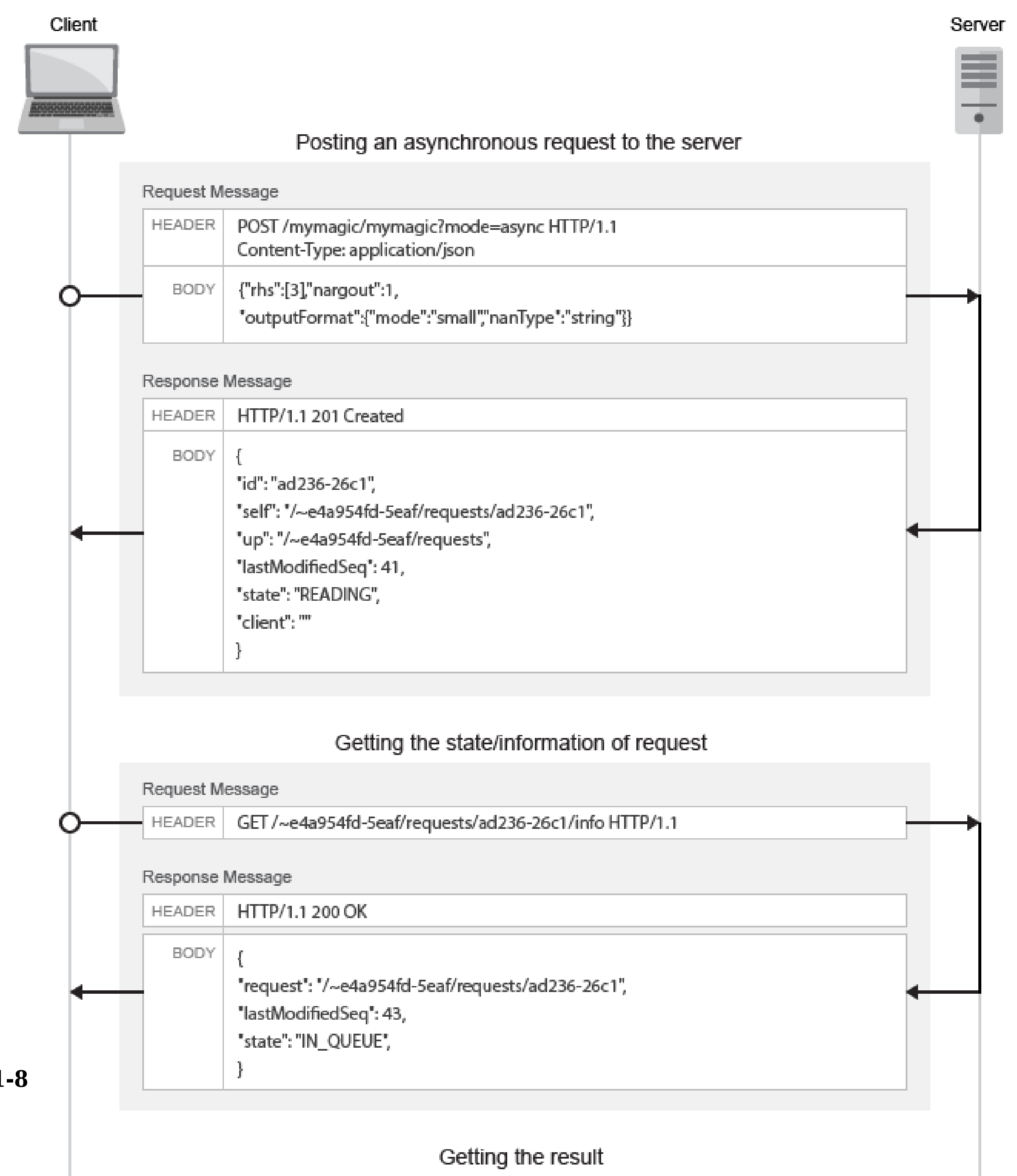
Asynchronous Execution

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
Client
Server
Posting an asynchronous request to the server
Request Message
HEADER
POST /mymagic/mymagic?mode=async HTTP/1.1
Content-Type: application/json
BODY
{"rhs":[3],"nargout":1,
"outputFormat":{"mode":"small","nanType":"string"}}
Response Message
HEADER
HTTP/1.1 201 Created
BODY
{
"id": "ad236-26c1",
"self": "/~e4a954fd-5eaf/requests/ad236-26c1",
"up": "/~e4a954fd-5eaf/requests",
"lastModifiedSeq": 41,
"state": "READING",
"client": ""
}
Getting the state/information of request
Request Message
HEADER
GET /~e4a954fd-5eaf/requests/ad236-26c1/info HTTP/1.1
Response Message
HEADER
HTTP/1.1 200 OK
BODY
{
"request": "/~e4a954fd-5eaf/requests/ad236-26c1",
"lastModifiedSeq": 43,
"state": "IN_QUEUE",
}
Getting the result

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” (MATLAB Production Server). For more information on setting up a server, see “Create a Server” (MATLAB Production 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>
        <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 + "&client=" + clien
            // 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/json and Specify HTTP
                contentType: 'application/json',
                method: 'POST',
                dataType: 'json',
                success: function(response) {
```

```
            // Print Request URI to webpage
            $("#requestURI").html('<strong>Request URI: </strong>' + hostname + response.self);
            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=" + request.id;
            $.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>' + requestResource.state);


                    if (requestResource.state != "READY" && requestResource.state != "ERROR") {
                        //Keep polling if the request is not done yet.
                        pollUsingUp(requestResource);
                    } else {
                        var requestURI = hostname + requestResource.self;
                        var responseURI = hostname + requestResource.self + "/result";
                        // Get result.
                        $.ajax({
                            url: responseURI,
                            // MPS RESTful API (Asynchronous): Specify HTTP GET method
                            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 large representation
                                    //response.lhs[0].mwdata + '</p>');

                                } else if (response.hasOwnProperty("error")) {
                                    alert("Error: " + response.error.message);
                                }
                                // MPS RESTful API (Asynchronous): Specify HTTP DELETE method
                                $.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>
```

Discovery Service

You can use the disovery service to learn about the MATLAB functions deployed to the server. The discovery service will return information about deployed MATLAB functions as a JSON object. The object is a multilevel nested structure and at a high level displays the discovery schema version and a list of deployed archives. Information about the deployed MATLAB functions and their function signatures is contained within each of the listed archives. For details, expand and see "JSON Response Object" (MATLAB Production Server).

In order to use the discovery service, you need to enable the discovery service on the server. You can do this by uncommenting the option `--enable-discovery` in the server's `main_config` file.

In order to get useful information when using the discovery service, you need to include a JSON file containing function signatures of the deployed MATLAB functions when creating the deployable archive. For more information, see "Package Deployable Archives with Production Server Compiler App" (MATLAB Production Server). For information about creating the JSON file containing function signatures, see "MATLAB Function Signatures in JSON" on page 1-17.

You can call the discovery service using the HTTP GET method.

RESTful API Calls for Discovery Service

Call	Purpose
GET Discovery Information	Discover MATLAB functions deployed on the server

The response from the server is a JSON object.

```
{
  "discoverySchemaVersion": "1.0.0",
    "archives": {

    "archive_name1": {
      "archiveSchemaVersion": "1.0.0",
      "archiveUuid": "uuid_string",

        "functions": {
        "matlab_function_name": {
          "signatures": [
            {
              "inputs": [
                { "name": "<param-name>", "mwtype": "<type-name>", "mwsize": [ "<size1>, ..., <sizeN>" ] },
                { "name": "<param-name>", "mwtype": "<type-name>", "mwsize": [ "<size1>, ..., <sizeN>" ] }
              ],
              "outputs": [
                { "name": "<param-name>", "mwtype": "<type-name>", "mwsize": [ "<size1>, ..., <sizeN>" ] },
                { "name": "<param-name>", "mwtype": "<type-name>", "mwsize": [ "<size1>, ..., <sizeN>" ] }
              ]
            }
          ]
        }
      },
      "matlabRuntimeVersion": "9.4.0",

      "typedefs": {

        "struct_name": {
          "type": "struct",
          "fields": [
            { "name": "field-name", "mwtype": "<type-name>", "mwsize": [ "<size1>, ..., <sizeN>" ] }
          ]
        },

        "homogeneous_cell_name": {
          "type": "cell",
          "elements": { "mwtype": "<element-type>", "mwsize": [ "<size1>, ..., <sizeN>" ] }
        },

        "heterogeneous_cell_name": {
          "type": "cell",
          "elements": [
            { "mwtype": "<element-type>", "mwsize": [ "<size1>, ..., <sizeN>" ] },
            { "mwtype": "<element-type>", "mwsize": [ "<size1>, ..., <sizeN>" ] }
          ]
        }
      }
    }
  }
}
```

JSON Response Object

`Object {;}`

Name	Description
`discoverySchemaVersion`	Version number for the discovery schema *Example value:* `1.0.0`
► `archives`	List of deployed archives. For details, see “Archives” (MATLAB Production Server)

Archives

`archives`

► `<archiveName>`

Name	Description
`archiveSchemaVersion`	Version number of the archive schema. *Example value:* `1.0.0`
`archiveUuid`	
► `functions`	A function object for each deployed function.For details, see “Functions” (MATLAB Production Server)
`matlabRuntimeVersion`	MATLAB Runtime version. *Example value:* `9.4.0`
► `typedefs`	If your deployed MATLAB functions require cell arrays or structs as inputs, they need to be defined as `typedefs`. For details, see “Typedefs” (MATLAB Production Server)

Functions

`functions`

► `<matlabFunctionName>`

  └►`signatures`

inputs

Name	Description
mwsize	Array size. *Example:* `"mwsize": ["2,3"]`
mwtype	MATLAB data type. *Example:* `"mwtype": "double"`
name	Name of input or output parameter. *Example:* `"name": "input1"`
typedef	Name of cell or struct. *Example:* `"typedef" : "mystruct'`

outpus

Name	Description
mwsize	Array size. *Example:* `"mwsize": ["2,3"]`
mwtype	MATLAB data type. *Example:* `"mwtype": "double"`
name	Name of input or output parameter. *Example:* `"name": "output1"`
typedef	Name of cell or struct. *Example:* `"typedef" : "mycell'`

Typedefs

typedefs

►<cellName>

elements

Name	Description
mwsize	Array size. *Example:* "mwsize": ["2,3"]
mwtype	Data type of element. *Example:* "mwtype": "double"

mwtype	cell

►<structName>

fields

Name	Description
mwsize	Array size. *Example:* "mwsize": ["2,3"]
mwtype	Data type of field value. *Example:* "mwtype": "char"
name	Field name. *Example:* "name": "my_field_name"

mwtype	struct

MATLAB Function Signatures in JSON

In order for a RESTful client to acquire the function signatures of MATLAB functions deployed to MATLAB Production Server using the discovery API, you need to embed information about your MATLAB functions in a JSON file while packaging your deployable archive.

After adding the MATLAB functions to deploy to the **Production Server Compiler** app, in the **Include MATLAB function signature file** section, select the **Create File** button. This action creates a template of the JSON file with the name `<*projectName*>functionSignatures.json`.

The `<*projectName*>functionSignatures.json` file is a single JSON object. It contains a schema version and a list of *function objects*. Each function object contains a list of *signature objects*, and each signature object contains a list of *argument objects*.

```
// Function Signatures
// To optionally specify argument types and/or sizes, search for "type"
// and insert the appropriate specifiers inside the brackets. For example:
//
//    "type": ["double", "size=1,1"]
//
// JSON-formatted text below this line.
{
    "_schemaVersion": "<major#>.<minor#>.<patch#>",
    "_typedefs":
    {
        "<struct_name>" :
        {
            "type" : "struct",
            "fields" :
            [
                {"name" : "<field_name>", "type" : "<field_matlab_type>"},
                {"name" : "<field_name>", "type" : "<field_matlab_type>"}
            ]
        },
        "<cell_name>" :
        {
            "type" : "cell",
            "elements" : { "type" : "element_matlab_type" }
        }
    },

    "functionName1":
    {
        "inputs":
        [
            { "name": "<input_name>", "type" : "<matlab_type>" },
            { "name": "<input_name>", "type" : ["<matlab_type>", "size=<array_dimensions>"] }
        ],

        "outputs":
        [
            {   "name": "<input_name>", "type" : "<matlab_type>" },
            {   "name": "<input_name>", "type" : ["<matlab_type>", "size=<array_dimensions>"] }
        ]
    },

    "functionName2":
    {
        "inputs":
        [
            { "name": "<input_name>", "type" : "<matlab_type>" },
            { "name": "<input_name>", "type" : ["<matlab_type>", "size=<array_dimensions>"] }
        ],

        "outputs":
        [
            {   "name": "<input_name>", "type" : "<matlab_type>" },
            {   "name": "<input_name>", "type" : ["<matlab_type>", "size=<array_dimensions>"] }
        ]
    }
}
```

Typedef object

Function object

Signature object

Argument objects

The schema version has a value that is a JSON string in the format <major#>.<minor#>.<patch#>, with each number specified as a nonnegative integer.

Function Objects

Function objects automatically inherit their name from the name of the MATLAB functions you've added to the project. The value of each function object is a signature object.

```
{
  "functionName1": { signatureObj1 },
  "functionName2": { signatureObj2 }
}
```

Signature Objects

A signature object defines the list of input and output arguments and supported platforms for the function. The value of the properties is an array of argument objects.

```
{
  "functionName1":
  {
     "inputs": [ argumentObj1, argumentObj2 ]
  }
}
```

Each signature can include the following properties.

Property	Description	JSON Data Type of Value
inputs	List of function input arguments	Array of argument objects
outputs	List of function output arguments	Array of argument objects

Argument Objects

Argument objects define the information for each of the input and output arguments.

```
{
  "functionName1":
  {
     "inputs":
```

```
      [
         {"name":"in1",  "type":["double"]},
         {"name":"in2",  "type":["logical"]}
      ]
   }
}
```

The order that the inputs appear in the JSON file is significant. For example, in a call to the `functionName1` function, `in1` must appear before `in2`.

Each argument object can include the following properties.

`name` — Name of argument

The name of the input or output argument, specified as a JSON string. This property and value are required. The `name` property does not need to match the argument name in the function, but it is a best practice for it to match any help documentation.

Example: `"name":"myArgumentName"`

`type` — Data type of argument

The `type` property defines what MATLAB data type the argument must have.

Value	Argument Description
`"double"`	Must be double precision number
`"single"`	Must be a single precision number
`"int8"`	Must be an 8-bit signed integer
`"uint8"`	Must be an 8-bit unsigned integer
`"int16"`	Must be a 16-bit signed integer
`"uint16"`	Must be a 16-bit unsigned integer
`"int32"`	Must be a 32-bit signed integer
`"uint32"`	Must be a 32-bit unisgned integer
`"int64"`	Must be a 64-bit signed integer
`"uint64"`	Must be a 64-bit unisgned integer
`"logical"`	Must be a logical array
`"char"`	Must be a character array

For `cell` and `struct`, see "Typedef Object" (MATLAB Production Server).

Example: `{ "name": "in", "type": ["double"] }`

`size` — Array dimensions

The `size` property defines the array dimensions of the inputs. It is a comma-seperated list of integers.

Example: `{ "name": "in", "type": ["double", "size=1,1" }`

Typedef Object

A typedef object defines cell arrays and structures. It needs to be defined only if any of the values to the argument objects are cells or structures. In the schema it is indicated using the name `_typedefs` and it's value is the name of one or more a cell or structure objects. The the type is the same as the argument object.

Example: If I have a MATLAB function called `sortinput` that accepts a cell array as input and returns a cell array as output, and each cell in the input consists of a structure, the JSON representation will be:

```
{
    "_schemaVersion": "1.0.0",
    "_typedefs" : {
        "struct_name": {
            "type": "struct",
            "fields": [
                {"name": "Name",  "type": "char"},
                {"name": "Score", "type": ["double","size=1,1"]}
            ]
        },
        "cell_name": {
            "type": "cell",
            "elements": {
                "type": "struct:struct_name"
            }
        }
    },
    "sortinput": {
        "inputs": [
            {
                "name": "unsorted_input",
```

```
                "type": ["cell:cell_name"]
            }
        ],
        "outputs": [
            {
                "name": "sorted_output",
                "type": ["cell:cell_name"]
            }
        ]
    }
}
```

Example: Heterogeneous cell with an array of type objects.

```
"<cell_name>" : {
            "type" : "cell",
            "elements" : [
                { "type" : ["<matlab_type>", "size=,array_dimmensions"] },
                { "type" : ["<matlab_type>", "size=,array_dimmensions"] },…
           ]
        }
```

See Also

More About

- “JSON Representation of MATLAB Data Types” (MATLAB Production Server)

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" (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` .

MATLAB Compiler SDK™ has four utility functions that make converting data between MATLAB and JSON simpler. They are:

Function Name	Purpose
`mps.json.encode` (MATLAB Compiler SDK)	Convert MATLAB data to JSON text using MATLAB Production Server JSON schema
`mps.json.decode` (MATLAB Compiler SDK)	Convert a character vector or string in MATLAB Production Server JSON schema to MATLAB data
`mps.json.encoderequest` (MATLAB Compiler SDK)	Convert MATLAB data in a server request to JSON text using MATLAB Production Server JSON schema
`mps.json.decoderesponse` (MATLAB Compiler SDK)	Convert JSON text from a server response to MATLAB data

JSON Representation of MATLAB Data Types

In this section...

"Numeric Types: double, single" on page 2-4
"Numeric Types: NaN, Inf, -Inf" on page 2-5
"Numeric Types: Integers" on page 2-6
"Numeric Types: Complex Numbers" on page 2-7
"Characters" on page 2-8
"Logical" on page 2-8
"Cell Arrays" on page 2-9
"Structures" on page 2-10
"Empty Arrays: []" on page 2-11
"Multidimensional Arrays" on page 2-11

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]`
		`}`

MATLAB Data Type	JSON Small Notation	JSON Large Notation

`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"}]`
		`}`

MATLAB Data Type	JSON Small Notation	JSON Large Notation

`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:	

MATLAB Data Type	JSON Large Notation

`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:		

MATLAB Data Type	JSON Small Notation	JSON Large Notation

`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:	

{'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": {
		"age": [
		{
		"mwdata": [15],
		"mwsize": [1,1],
		"mwtype": "double"
		}
		],
		"name": [
		{
		"mwdata": ["John Smith"],
		"mwsize": [1,10],
		"mwtype": "char"
		}
		]
		}
		}

<table>
<tr><th>MATLAB Data Type</th><th>JSON Small Notation</th><th>JSON Large Notation</th></tr>
<tr><td>x = struct('Name',{{'Casper','Ghost'}},...
'Age',{[14,17,18]},...
'Date', {736676});</td><td>No small representation.</td><td>{
"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"}]}
}</td></tr>
</table>

Empty Arrays: []

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

<table>
<tr><th>MATLAB Data Type</th><th>JSON Small Notation</th><th>JSON Large Notation</th></tr>
<tr><td>[]</td><td>[]</td><td>{
"mwtype": "double" | "single"
"int8" | "uint8" |
"int32" | "uint32" |
"logical" | "char" |
"mwsize": [0,0],
"mwdata": [ ]
}</td></tr>
</table>

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]`
	`             }`
	`}`

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

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},
     ...]}}
```

Structure of HTTP Error

```
{
    "error": {
        "type": "httperror",
        "code": 404,
        "messageId": "ComponentNotFound",
        "message": "Component not found."
    }
}
```

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-All
```

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

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 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 4-4
“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 https://www.mathworks.com/products/compiler/mcr. See “Download and Install the MATLAB Runtime” (MATLAB Production Server) for more information.
2. Create a server using `mps-new`. See “Create a Server” (MATLAB Production 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” (MATLAB Production Server) for details.
4. Start the server using `mps-start` (MATLAB Production Server), and verify it is running with `mps-status` (MATLAB Production Server).
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" (MATLAB Production Server) 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

C: Coupon Payment

N: Number of payments

i: Interest rate

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

POST Synchronous Request

Make a synchronous request to the server, and wait for a response

Description

Use a POST method to make a synchronous request to the server. In synchronous mode, once a request has been made, the server blocks all further requests until it has completed processing the original request. A response is automatically returned once processing is complete. No other HTTP methods are necessary to retrieve the response from the server.

Request

HTTP Method

`POST`

URI

`http://host:port/deployedArchiveName/matlabFunctionName`

Query Parameters

None.

Content-Type

`application/json`

Body

Name	Description	Value-Type
`nargout`	Number of outputs that the client application is requesting from the deployed MATLAB function. Note that MATLAB functions, depending on their intended purpose, can be coded to return multiple outputs. A subset of these potential outputs can be specified using `nargout`.	`number`
`rhs`	Input arguments to the deployed MATLAB function, specified as an array of comma-separated values.	`[arg1,arg2,arg3,...]`
`outputFormat`	Specify whether the MATLAB output in the response should be returned using large or small JSON representation, and whether `NaN` and `Inf` should be represented as a JSON string or object.	`{ "mode" : "small | large", "nanInfFormat" : "string | object" }`

Example:

Single Input Argument:

```
{
 "nargout": 1,
 "rhs": [5],
 "outputFormat" : { "mode" : "small","nanInfFormat": "object"}
}
```

Multiple Input Arguments:

```
{
 "nargout": 2,
 "rhs": [3, 4, 5 ...],
 "outputFormat" : { "mode" : large", "nanInfFormat" : "string" }
}
```

Response

Success

`200 OK`

Name	Description	Value-Type
`lhs`	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 notation. For more information on JSON notation see "JSON Representation of MATLAB Data Types" on page 2-2.	`[output1, output2, ...]`

Example:


```
{
"lhs":[[[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]]]
}
```


Error

`400 InvalidJSON`

`404 FunctionNotFound`

`404 ComponentNotFound`

Sample Call

HTTP

Request:

```
POST /mymagic/mymagic HTTP/1.1
Host: localhost:9910
Content-Type: application/json

{"rhs":[5],"nargout":1,"outputFormat":{"mode":"small","nanType":"string"}}
```

Response:

```
Status Code: 200 OK
{
"lhs":[[[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]]]
}
```

JavaScript

```
var data = JSON.stringify({
    "rhs": [5],
    "nargout": 1,
    "outputFormat": {"mode": "small", "nanType": "string"}
});
var xhr = new XMLHttpRequest();
xhr.addEventListener("readystatechange", function () {
    if (this.readyState === 4) {
        console.log(this.responseText);
    }
});
xhr.open("POST", "http://localhost:9910/mymagic/mymagic");
xhr.setRequestHeader("content-type", "application/json");
xhr.send(data);
```

See Also

Topics

“Synchronous Execution” on page 1-2
“Example: Synchronous Execution of Magic Square using RESTful API and JSON” on page 1-4

Introduced in R2016a

POST Asynchronous Request

Make an asynchronous request to the server

Description

Use a POST method to make an asynchronous request to the server. During asynchronous execution, this step is usually the first in the process.

Request

HTTP Method

`POST`

URI

`http://host:port/deployedArchiveName/matlabFunctionName`

Query Parameters

Name	Description	Value-Type
`mode`	(Required). Specify mode of communication.	`async`
`client`	(Optional). Specify an ID or name for the client making the request.	`{client-id-string}`

Example:

`?mode=async&client=Nor101`

Content-Type

`application/json`

Body

Name	Description	Value-Type
`nargout`	Number of outputs that the client application is requesting from the deployed MATLAB function. Note that MATLAB functions, depending on their intended purpose, can be coded to return multiple outputs. A subset of these potential outputs can be specified using `nargout`.	`number`
`rhs`	Input arguments to the deployed MATLAB function, specified as an array of comma-separated values.	`[arg1,arg2,arg3,...]`
`outputFormat`	Specify whether the MATLAB output in the response should be returned using large or small JSON representation, and whether `NaN` and `Inf` should be represented as a JSON string or object.	`{ "mode" : "small | large", "nanInfFormat" : "string | object" }`

Example:

Single Input Argument:

```
{
 "nargout": 1,
 "rhs": [5],
 "outputFormat" : { "mode" : "small,"nanInfFormat": "object"}
}
```

Multiple Input Arguments and Multiple Outputs:

```
{
 "nargout": 2,
 "rhs": [3, 4, 5 ...],
 "outputFormat" : { "mode" : large", "nanInfFormat" : "string" }
}
```

Response

Success

`201 Created`

Name	Description	Value-Type
`id`	ID of a particular request.	`{id-string}`
`self`	URI of particular request.	`{request-uri-string}`
`up`	URI of a collection of requests tied to a particular client.	`{request-collection-uri-string}`
`lastModifiedSeq`	Number indicating when a request represented by `self` was last modified.	`{server-state-number}`

`state`	State of a request.	`{request-state-string}`
		List of states:
		`READING`
		`IN_QUEUE`
		`PROCESSING`
		`READY`
		`ERROR`
		`CANCELLED`

`client`	Client id/name that was specified as a query parameter while initiating a request.	`{client-id-string}`

Example:

```
{
  "id": "a061c723-4724-42a0-b405-329cb8c373d6",
  "self": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/a061c723-4724-42a0-b405-329
  "up": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests",
  "lastModifiedSeq": 6,
  "state": "READING",
  "client": ""
}
```


Error

`404 ResourceNotFound`

`405 MethodNotAllowed` — No `'Access-Control-Allow-Origin'` header. Enable CORS on server.

`415 InvalidContentType`

`415 UnsupportedMediaType`

Sample Call

HTTP

Request:

```
POST /mymagic/mymagic?mode=async HTTP/1.1
Host: localhost:9910
Content-Type: application/json

{"rhs":[7],"nargout":1,"outputFormat":{"mode":"small","nanType":"string"}}
```

Response:

```
Status Code: 201 Created
Header:
  Location: /~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/ad2363f3-26c1-4d48-88f8-6b
  X-MPS-Start-Time: 003472d705bd1cd2
  Content-Length: 248
Body:
{
  "id": "ad2363f3-26c1-4d48-88f8-6b7fb615f254",
  "self": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/ad2363f3-26c1-4d48-88f8-6b7
  "up": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests",
  "lastModifiedSeq": 41,
  "state": "READING",
  "client": ""
}
```

JavaScript

```
var data = JSON.stringify(
    {   "rhs": [7],
        "nargout": 1,
        "outputFormat": {"mode": "small","nanType": "string"}
    }
);

var xhr = new XMLHttpRequest();
xhr.open("POST", "http://localhost:9910/mymagic/mymagic?mode=async");
xhr.setRequestHeader("content-type", "application/json");
xhr.addEventListener("readystatechange", function () {
    if (this.readyState === 4) {
        console.log(this.responseText);
    }
});
xhr.send(data);
```

See Also

GET Representation of Asynchronous Request | POST Cancel Request | DELETE Request

Topics

“Asynchronous Execution” on page 1-6
“Example: Asynchronous Execution of Magic Square using RESTful API and JSON” on page 1-9
“Example: Web-Based Bond Pricing Tool Using JavaScript” on page 4-2

Introduced in R2016b

GET Representation of Asynchronous Request

View how an asynchronous request made to the server is represented

Description

Use a GET method to view the representation of an asynchronous request on the server. The URI of the `self` field serves as the addressable resource for the method.

Request

HTTP Method

```
GET
```

URI

```
http://host:port/{request-uri-string}
```

Response

Success

```
200 OK
```

Name	Description	Value-Type
`id`	ID of a particular request.	`{id-string}`
`self`	URI of particular request.	`{request-uri-string}`

Name	Description	Value-Type
`up`	URI of a collection of requests tied to a particular client.	`{request-collection-uri-string}`
`lastModifiedSeq`	Number indicating when a request represented by `self` was last modified.	`{server-state-number}`

`state`	State of a request.	`{request-state-string}`
		
		Possible states:
		
		`READING`
		`IN_QUEUE`
		`PROCESSING`
		`READY`
		`ERROR`
		`CANCELLED`

`client`	Client id or name that was specified as a query parameter while initiating an asynchronous request.	`{client-id-string}`

Example:

```
{
  "id": "f90c2ff8-4d27-4795-806d-18c351abeb5b",
  "self": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/f90c2ff8-4d27-4795-806d-18c
  "up": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests",
  "lastModifiedSeq": 30,
  "state": "READING",
  "client": "786"
}
```

Error

`400 NoMatchForQueryParams`

`404 ResourceNotFound`

Sample Call

HTTP

Request:

```
GET /~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/f90c2ff8-4d27-4795-806d-18c351abeb
Host: localhost:9910
```

Response:

```
Status Code: 200 OK
{
  "id": "f90c2ff8-4d27-4795-806d-18c351abeb5b",
  "self": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/f90c2ff8-4d27-4795-806d-18c
  "up": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests",
  "lastModifiedSeq": 31,
  "state": "IN_QUEUE",
  "client": "786"
}
```

JavaScript

```
var data = null;
var xhr = new XMLHttpRequest();
xhr.addEventListener("readystatechange", function () {
    if (this.readyState === 4) {
        console.log(this.responseText);
    }
});
xhr.open("GET", "http://localhost:9910/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/
xhr.send(data);
```

See Also

GET State Information | GET Result of Request

Introduced in R2016b

GET Collection of Requests

View a collection of requests

Description

Use a GET method to view a collection of requests on the server. The URI of the up field serves as the addressable resource for the method.

Request

HTTP Method

GET

URI

http://host:port/{request-collection-uri-string}

Query Parameters

Name	Description	Value-Type
since	Required.	{server-state-number}
clients	Required if ids is not specified.	{client-id-string_1}, {client-id-string_2},...
ids	Required if clients is not specified.	{id-string_1}, {id_string_2},...

Example:

?since=30&clients=786

- The query parameter `since={server-state-number}` is *required* if you are making an asynchronous request.
- The query parameter `clients={client-id-string}` is *optional*.

Response

Success

`200 OK`

Name	Description	Value-Type
`createdSeq`	Number indicating the server state. The requests included in the `data` collection are the requests that have gone through some state change between `since` and `createdSeq`.	`{server-state-number}`

`data`	Collection of MATLAB execution requests that match a query.	`"data": [`
		`{`
		`"id": {id-string},`
		`"self": [request-uri-string},`
		`"up": {request-collection-uri-string},`
		`"lastModifiedSeq": {server-state-numbe`
		`"state": {request-state-string},`
		`"client": {client-id-string}`
		`},`
		`{`
		`"id": {id-string},`
		`"self": {request-uri-string},`
		`"up": {request-collection-uri-string},`
		`"lastModifiedSeq": {server-state-numbe`
		`"state": {request-state-string},`
		`"client": {client-id-string}`
		`},...`
		`]`

Example:

```
"data": [
    {
      "id": "c5666088-b087-4bae-aa7d-d8470e6e082d",
      "self": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/c5666088-b087-4bae-aa7d
      "up": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests",
      "lastModifiedSeq": 19,
      "state": "READY",
      "client": "786"
    },
    {
      "id": "a4d0f902-d212-47d5-a855-6d64192842d8",
      "self": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/a4d0f902-d212-47d5-a855
      "up": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests",
      "lastModifiedSeq": 17,
      "state": "READY",
      "client": "786"
    },

  ]
```


Error

400 InvalidParamSince

400 MissingParamSince

400 MissingQueryParams

400 NoMatchForQueryParams

404 URL not found

500 InternalServerError

Sample Call

HTTP

Request:

```
GET /~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests?since=15&clients=786 HTTP/1.1
Host: localhost:9910
```

Response:

```
Status Code: 200 OK
{
  "createdSeq": 19,
  "data": [
    {
      "id": "c5666088-b087-4bae-aa7d-d8470e6e082d",
      "self": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/c5666088-b087-4bae-aa7d
      "up": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests",
      "lastModifiedSeq": 19,
      "state": "READY",
      "client": "786"
    },
    {
      "id": "a4d0f902-d212-47d5-a855-6d64192842d8",
      "self": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/a4d0f902-d212-47d5-a855
      "up": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests",
      "lastModifiedSeq": 17,
      "state": "READY",
      "client": "786"
    }
   ]
  }
```

JavaScript

```
var data = null;
var xhr = new XMLHttpRequest();
xhr.addEventListener("readystatechange", function () {
    if (this.readyState === 4) {
        console.log(this.responseText);
    }
});
xhr.open("GET", "http:////localhost:9910/~e4a954fd-5eaf-4b54-aac2-20681b33d075/request
xhr.send(data);
```

See Also

GET State Information | GET Representation of Asynchronous Request

Introduced in R2016b

GET State Information

Get state information of request

Description

Use a GET method to get information about the state of a request. The URI of the `self` field serves as the addressable resource for the method. Possible states are: `READING`, `IN_QUEUE`, `PROCESSING`, `READY`, `ERROR`, and `CANCELLED`.

Request

HTTP Method

`GET`

URI

`http://host:port/{request-uri-string}/info`

Response

Success

`200 OK`

Name	Description	Value-Type
`request`	URI to current request.	`{request-uri-string}`
`lastModifiedSeq`	Number indicating when the current request was last modified.	`{server-state-number}`

Name	Description	Value-Type
state	State of current request.	{request-state-string} Possible states: READING IN_QUEUE PROCESSING READY ERROR CANCELLED

Example:

```
{
  "request": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/c5666088-b087-4bae-aa7d-
  "lastModifiedSeq": 19,
  "state": "READY"
}
```

Error

400 NoMatchForQueryParams— Query with invalid request ID.

404 URL not found

Sample Call

HTTP

Request:

```
GET /~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/c5666088-b087-4bae-aa7d-d8470e6e08
Host: localhost
Port: 9910
```

Response:

```
Status Code: 200 OK
{
  "request": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/c5666088-b087-4bae-aa7d-
  "lastModifiedSeq": 19,
  "state": "READY"
}
```

JavaScript

```
var data = null;
var xhr = new XMLHttpRequest();
xhr.addEventListener("readystatechange", function () {
    if (this.readyState === 4) {
        console.log(this.responseText);
    }
});
xhr.open("GET", "http://localhost:9910/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/
xhr.send(data);
```

See Also

GET Representation of Asynchronous Request | GET Result of Request

Introduced in R2016b

GET Result of Request

Retrieve the results of a request

Description

Use a GET method to retrieve the results of a request from the server. The URI of the `self` field serves as the addressable resource for the method.

Request

HTTP Method

`GET`

URI

`http://host:port/{request-uri-string}/result`

Response

Success

`200 OK`

Results represented in JSON.

Example:


```
{"lhs":[[[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]]]}
```

Error

404 RequestNotFound

410 RequestAlreadyCompleted

410 RequestAlreadyCancelled

410 RequestAlreadyDeleted

500 InternalServerError

Sample Call

HTTP

Request:

```
GET /~f76280c5-b94c-4cd9-8eb6-841532788583/requests/ad063314-ebda-4310-b356-59420058c1
Host: localhost:9910
```

Response:

```
Status Code: 200 OK
{"lhs":[[[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]]]
```

JavaScript

```
var data = null;
var xhr = new XMLHttpRequest();
xhr.addEventListener("readystatechange", function () {
  if (this.readyState === 4) {
    console.log(this.responseText);
  }
});
xhr.open("GET", "http://localhost:9910/~f76280c5-b94c-4cd9-8eb6-841532788583/requests/
xhr.send(data);
```

See Also

GET State Information | DELETE Request

Introduced in R2016b

POST Cancel Request

Cancel a request

Description

Use a POST method to cancel a request. Only requests that have not already completed can be canceled.

Request

HTTP Method

```
POST
```

URI

```
http://host:port/{request-uri-string}/cancel
```

Response

Success

```
204 No Content
```

Error

```
404 RequestNotFound
```

```
410 RequestAlreadyCompleted
```

```
410 RequestAlreadyCancelled
```

410 RequestAlreadyDeleted

500 InternalServerError

Sample Call

HTTP

Request:

```
POST /~f76280c5-b94c-4cd9-8eb6-841532788583/requests/ef90fca4-0d3c-4395-8dc8-af8a8905b
Host: localhost:9910
```

Response:

```
Status Code: 204 No Content
```

JavaScript

```
var data = null;
var xhr = new XMLHttpRequest();
xhr.addEventListener("readystatechange", function () {
  if (this.readyState === 4) {
    console.log(this.responseText);
  }
});
xhr.open("POST", "http://localhost:9910/~f76280c5-b94c-4cd9-8eb6-841532788583/requests
xhr.send(data);
```

See Also

DELETE Request | POST Asynchronous Request

Introduced in R2016b

DELETE Request

Delete a request

Description

Use a DELETE method to delete a request on the server. Once a request has been deleted, information about it cannot be retrieved.

Request

HTTP Method

`DELETE`

URI

`http://host:port/{request-uri-string}`

Response

Success

`204 No Content`

Error

`404 RequestNotFound`

`409 RequestNotCompleted`— Request has not reached terminal state.

`410 RequestAlreadyDeleted`

```
500 InternalServerError
```

Sample Call

HTTP

Request:

```
DELETE /~f76280c5-b94c-4cd9-8eb6-841532788583/requests/31577b58-209c-4c41-b3f8-6e1e025
Host: localhost:9910
```

Response:

```
Status Code: 204 No Content
```

JavaScript

```
var data = null;
var xhr = new XMLHttpRequest();
xhr.addEventListener("readystatechange", function () {
  if (this.readyState === 4) {
    console.log(this.responseText);
  }
});

xhr.open("DELETE", "http://localhost:9910/~f76280c5-b94c-4cd9-8eb6-841532788583/reques
xhr.send(data);
```

See Also

POST Cancel Request | POST Asynchronous Request

Introduced in R2016b

GET Discovery Information

Discover MATLAB functions deployed on the server

Description

Use the GET method to view information about the MATLAB functions deployed to the server. You receive information about:

- all deployed archives with discovery information
- names of the MATLAB functions contained in each of the archives
- names and MATLAB data types of the inputs and outputs for each of the MATLAB functions
- additional metadata

If a deployable archive (`.ctf` file) has been built without including discovery information, it is not discoverable.

In order to use the discovery service, you need to enable the discovery service on the server. You can do this by uncommenting the option `--enable-discovery` in the server's `main_config` file.

Request

HTTP Method

`GET`

URI

`http://host:port/api/discovery`

Response

Success

```
200 OK
```

For a description of the body, see "JSON Response Object" on page 1-14.

Error

```
403 DiscoveryDisabled
```

Sample Call

HTTP

Request:

```
GET /api/discovery HTTP/1.1
Host: localhost:9910
```

Response:

```
{
    "discoverySchemaVersion": "1.0.0",
    "archives": {
        "mymagic": {
            "archiveSchemaVersion": "1.0.0",
            "archiveUuid": "mymagic_73BCCE8B5FFFB984888169285CBA8A31",
            "functions": {
                "mymagic": {
                    "signatures": [
                        {
                            "inputs": [
                                {
                                    "mwsize": [],
                                    "mwtype": "double",
                                    "name": "in"
                                }
                            ],
                            "outputs": [
                                {
                                    "mwsize": [],
                                    "mwtype": "double",
                                    "name": "out"
                                }
                            ]
                        }
                    ]
                }
            },
            "matlabRuntimeVersion": "9.4.0"
        }
```

```
    }
}
```

JavaScript

```
var data = null;
var xhr = new XMLHttpRequest();
xhr.addEventListener("readystatechange", function () {
  if (this.readyState === 4) {
    console.log(this.responseText);
  }
});

xhr.open("GET", "http://localhost:9910/api/discovery");
xhr.send(data);
```

See Also

Topics

"Discovery Service" on page 1-11
"MATLAB Function Signatures in JSON" on page 1-17

Introduced in R2018a

6

Functions — Alphabetical List

mps.json.encode

Convert MATLAB data to JSON text using MATLAB Production Server JSON schema

Syntax

```
text = mps.json.encode(data)
text = mps.json.encode(data,Name,Value)
```

Description

`text = mps.json.encode(data)` encodes MATLAB data and returns JSON text in JSON schema for MATLAB Production Server. You can use this JSON text on multiple platforms to encode content for MATLAB Production Server.

`text = mps.json.encode(data,Name,Value)` specifies additional options with one or more name-value pair arguments for specific input cases. For example, you can decide to encode `data` in the large or small format defined for representing data types.

Examples

Convert a Matrix to JSON Schema for MATLAB Production Server

Encode a 3-by-3 magic square in the JSON format.

```
mps.json.encode(magic(3))
```

```
ans =
    '[[8,1,6],[3,5,7],[4,9,2]]'
```

Convert a Matrix and Specify Format for JSON Schema for MATLAB Production Server

Encode a 3-by-3 magic square in JSON using the `large` format option.

```
mps.json.encode(magic(3),'Format','large')

ans =
    '{"mwdata":[8,3,4,1,5,9,6,7,2],"mwsize":[3,3],"mwtype":"double"}'
```

Convert an Array Containing NaN, Inf, or -Inf to JSON Schema for MATLAB Production Server

Encode an array containing `-Inf`, `NaN`, and `Inf` in JSON using '`object`' in '`NanInfType`' option.

```
mps.json.encode([-Inf NaN Inf],'NaNInfType','object','Format','large')

ans =
    '{"mwdata":[{"mwdata":"-Inf"},{"mwdata":"NaN"},{"mwdata":"Inf"}], "mwsize":[1,3],"mwtype":"double"}'
```

Input Arguments

data — MATLAB data that MATLAB Production Server supports
numeric | character | logical | structure | cell

MATLAB data that MATLAB Production Server supports, specified as a numeric, character, logical, structure, or cell.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `mps.json.encode(data,'Format','large')`

Format — Format to encode data
'small' (default) | 'large'

Format to encode MATLAB data, specified as the comma-separated pair consisting of '`Format`' and the format '`small`' or '`large`'.

The `small` format is a more simple representation of MATLAB data types in JSON, whereas the `large` format is a more generic representation. For more information, see “JSON Representation of MATLAB Data Types” on page 2-2.

`NaNInfType` — Format to encode `NaN`, `Inf`, and `-Inf` in `data`
'string' (default) | 'object'

Format to encode `NaN`, `Inf`, and `-Inf` in `data`, specified as a comma-separated pair consisting of '`NaNInfType`' and the JSON data-types '`string`' or '`object`'.

`PrettyPrint` — Format `text` for readability
false (default) | true

Format `text` for readability, specified as a comma-separated pair consisting of '`PrettyPrint`' and logical '`true`' or '`false`'.

`PrettyPrint` enables better readability for a user when set to `true`. Syntax is `mps.json.encode(magic(3),'PrettyPrint',true)`.

Output Arguments

`text` — JSON-formatted text
character vector

JSON-formatted text for JSON schema for MATLAB Production Server, returned as a character vector.

See Also

`mps.json.decode` | `mps.json.encoderequest` | `mps.json.decoderesponse`

Introduced in R2018a

mps.json.decode

Convert a character vector or string in MATLAB Production Server JSON schema to MATLAB data

Syntax

```
data = mps.json.decode(text)
```

Description

`data = mps.json.decode(text)` parses JSON schema for MATLAB Production Server to convert it to MATLAB data.

Examples

Decode JSON-Formatted Text for a Matrix

```
mps.json.decode('[[8,1,6],[3,5,7],[4,9,2]]')
```

```
ans =
     8     1     6
     3     5     7
     4     9     2
```

Decode a Matrix in JSON That Uses `large` Format

```
mps.json.decode('{"mwdata":[1,4,3,2],"mwsize":[2,2],"mwtype":"double"}')
```

```
ans =
       1     3
       4     2
```

Input Arguments

`text` — JSON text following the schema for MATLAB Production Server
character vector (default) | string

JSON following the schema for MATLAB Production Server, specified as a character vector or string.

`text` can be in various formats like `small`, `large`, `NaNInfType`, and `PrettyPrint`, as explained in "Name-Value Pair Arguments" on page 6-3 on the `mps.json.encode` page.

Output Arguments

`data` — MATLAB data
any MATLAB data type

MATLAB data decoded from MATLAB Production Server JSON text returned as the data-type encoded in `text`.

See Also

`mps.json.encode` | `mps.json.encoderequest` | `mps.json.decoderesponse`

Introduced in R2018a

mps.json.encoderequest

Convert MATLAB data in a server request to JSON text using MATLAB Production Server JSON schema

Syntax

```
text = mps.json.encoderequest(rhs)
text = mps.json.encoderequest(rhs,Name,Value)
```

Description

`text = mps.json.encoderequest(rhs)` encodes the request that is input to the deployed MATLAB function using JSON schema for MATLAB Production Server. It builds a server request that includes MATLAB variables and options such as “'Nargout'” on page 6-0 and “'OutputFormat'” on page 6-0 , that are needed to make a call to the MATLAB Production Server.

`text = mps.json.encoderequest(rhs,Name,Value)` specifies additional options with one or more name-value pair arguments for specific input cases.

Examples

Write MATLAB Production Server Payload

```
mps.json.encoderequest({[1 2 3 4]})
```

```
ans =
    '{"rhs":[[[1,2,3,4]]],"nargout":1,"outputFormat":{"mode":"small","nanType":"string"}}'
```

Write MATLAB Production Server Payload, and Set Output Parameters

For `rhs = {['Red'], ['Blue'], ['Yellow'], ['Green']}`

```
mps.json.encoderequest(rhs, 'Nargout', 3, 'OutputFormat', 'large')

ans =
    '{"rhs":["Red","Blue","Yellow","Green"],"nargout":3,"outputFormat":{"mode":"large","nanType":"string"}}'
```

Input Arguments

`rhs` — Input arguments for deployed MATLAB function that is called

cell vector of any MATLAB data type supported by MATLAB Production Server

Input arguments for a MATLAB function deployed on MATLAB Production Server that is called, specified as a cell vector.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `mps.json.encoderequest(rhs, 'Format', 'large')`

`Nargout` — Number of output arguments for function deployed on MATLAB Production Server

1 (default) | any positive integer

Number of output arguments for function deployed on MATLAB Production Server specified as comma-separated pair consisting of '`Nargout`' and number of output arguments.

`mps.json.encoderequest(rhs, 'Nargout', 3)`.

`Format` — Format to encode `rhs`

'small' (default) | 'large'

Format to encode `rhs` specified as comma-separated pair consisting of '`Format`' and the format '`small`' or '`large`'.

The `small` format is a more simple representation of MATLAB data types in JSON, whereas the `large` format is a more generic representation. For more information, see “JSON Representation of MATLAB Data Types” on page 2-2.

`NaNInfType` — Format to encode `NaN`, `Inf`, `-Inf` in `rhs`
'string' (default) | 'object'

Format to encode `NaN`, `Inf`, `-Inf` in `rhs` specified as comma-separated pair consisting of '`NaNInfType`' and JSON data-types '`string`' and '`object`'.

`OutputFormat` — Format for response from MATLAB function deployed on MATLAB Production Server
'small' (default) | 'large'

Format for response from MATLAB function deployed on MATLAB Production Server specified as comma-separated pair consisting of '`OutputFormat`' and the format '`small`' or '`large`'.

Output format is set using `mps.json.encoderequest(rhs, 'OutputFormat', 'large')`.

`OutputNanInfType` — Type for response from MATLAB function deployed on MATLAB Production Server containing `NaN`, `Inf`, `-Inf`
'string' (default) | 'object'

Type for response from MATLAB function deployed on MATLAB Production Server containing `NaN`, `Inf`, `-Inf` specified as comma-separated pair consisting of '`OutputNaNInfType`' and JSON data-types '`string`' and '`object`'.

NaN-type for output response is set using `mps.json.encoderequest(rhs, 'OutputNaNInfType', 'object')`.

`PrettyPrint` — Format text for readability
false (default) | true

Format `text` for readability specified as a comma-separated pair consisting of '`PrettyPrint`' and logical '`true`' or '`false`'. Syntax is `mps.json.encoderequest(rhs,'PrettyPrint',true)`.

Output Arguments

`text` — JSON text
character vector

JSON-formatted text for JSON schema for MATLAB Production Server, returned as a character vector.

See Also

`mps.json.encode` | `mps.json.decode` | `mps.json.decoderesponse`

Introduced in R2018a

mps.json.decoderesponse

Convert JSON text from a server response to MATLAB data

Syntax

```
lhs = mps.json.decoderesponse(response)
error = mps.json.decoderesponse(response)
```

Description

`lhs = mps.json.decoderesponse(response)` reads the JSON payload of the output arguments returned from a successful MATLAB function call.

`error = mps.json.decoderesponse(response)` reads the JSON payload of the MATLAB error thrown from a failed MATLAB function call.

Examples

Read from MATLAB Production Server Payload

```
mps.json.decoderesponse('{"lhs":[[[1, 2, 3, 4]]]}')
```

```
ans =
  1x1 cell array
    {1x4 double}
```

Read response from a `sortstudent` function deployed on MATLAB Production Server

This example illustrates the execution of `mps.json.encoderequest` and `mps.json.decoderesponse` to call a function deployed on MATLAB Production Server using `webwrite`. In this case, student names and their corresponding scores are deployed to MATLAB Production Server to the `sortstudents` function that sorts students based on their scores. The result returned is the equivalent to calling the

function `sortstudents(struct('name', 'Ed', 'score', 83), struct('name', 'Toni', 'score', 91))` from MATLAB.

```
data = {struct('name', 'Ed', 'score', 83), struct('name', 'Toni', 'score', 91)};
body = mps.json.encoderequest(data);

options = weboptions;

% Create a weboptions object that instructs webread to return JSON text
options.ContentType = 'text';

% Create a weboptions object that instructs webwrite to encode character vector data as JSON to post it to a web service
options.MediaType = 'application/json';

response = webwrite('http://localhost:9910/studentapp/sortstudents', body, options);

result = mps.json.decoderesponse(response);
```

Input Arguments

response — JSON result from a MATLAB function call
char (default)

JSON result from a MATLAB function call specified as JSON text.

Output Arguments

`lhs` — Cell vector of output arguments
`Cell` vector

Cell vector of output arguments that are from a MATLAB function called from MATLAB Production Server.

`error` — Generated output when request results in a MATLAB error
`struct` array

Generated output when request to MATLAB function called from MATLAB Production Server results in a MATLAB error returned as a struct array.

See Also

`mps.json.encode` | `mps.json.decode` | `mps.json.encoderequest`

Introduced in R2018a