

MATLAB® Production Server™

RESTful API and JSON



MATLAB®

R2020b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

MATLAB® Production Server™ RESTful API and JSON

© COPYRIGHT 2016–2020 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 2017a) |
| September 2017 | Online only | Revised for Version 3.0.1 (Release R2017b) |
| March 2018 | Online only | Revised for Version 3.1 (Release R2018a) |
| September 2018 | Online only | Revised for Version 4.0 (Release R2018b) |
| March 2019 | Online only | Revised for Version 4.1 (Release R2019a) |
| September 2019 | Online only | Revised for Version 4.2 (Release R2019b) |
| March 2020 | Online only | Revised for Version 4.3 (Release R2020a) |
| September 2020 | Online only | Revised for Version 4.4 (Release R2020b) |

Client Programming

1

| | |
|--|-------------|
| RESTful API | 1-2 |
| Synchronous Execution | 1-3 |
| Example: Synchronous Execution of Magic Square Using RESTful API and JSON | 1-3 |
| Asynchronous Execution | 1-5 |
| Example: Asynchronous Execution of Magic Square Using RESTful API and JSON | 1-8 |
| Asynchronous Request Execution on Azure Using HTTPS | 1-9 |
| Discovery Service | 1-10 |
| Health Check | 1-14 |
| | |
| MATLAB Function Signatures in JSON | 1-16 |
| Function Objects | 1-17 |
| Signature Objects | 1-18 |
| Argument Objects | 1-18 |
| Typedef Object | 1-19 |

JSON Representation of MATLAB Data Types

2

| | |
|---|-------------|
| JSON Representation of MATLAB Data Types | 2-2 |
| Numeric Types: double, single and Integers | 2-3 |
| Numeric Types: NaN, Inf, and -Inf | 2-5 |
| Numeric Types: Complex Numbers | 2-6 |
| Character Array | 2-7 |
| Logical | 2-8 |
| Cell Array | 2-9 |
| Structure Array | 2-10 |
| String Array | 2-12 |
| Enumeration | 2-13 |
| Datetime Array | 2-14 |
| Empty Array: [] | 2-19 |

Troubleshooting RESTful API Errors

3

| | |
|---|------------|
| Troubleshooting RESTful API Errors | 3-2 |
| Structure of MATLAB Error | 3-3 |

| | |
|-----------------------------------|-----|
| Structure of HTTP Error | 3-4 |
| Access-Control-Allow-Origin | 3-4 |

Examples: RESTful API and JSON

4

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

RESTful APIs

5

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.
- Unique Uniform Resource Identifiers (URIs) identify the resources that the server creates.
- Message headers convey metadata such as the *Content-Type* of a request.
 - The API supports `application/json` as the HTTP *Content-Type* header.
 - The RESTful API for MATLAB function execution also supports `application/x-google-protobuf` as the HTTP *Content-Type* through the Java® and .NET client APIs only.
- The message body of the request contains information to be sent to the server.
 - If you use JSON as the data serialization format, inputs to the MATLAB function contained within a deployed archive are represented in JSON and encapsulated within the body of a message.
 - If you use protocol buffers (protobuf) for data serialization, the Java and .NET client libraries provide helper classes to internally create protobuf messages based on a proto format and returns the corresponding byte array. Use this byte array in the message body of the request.
- The message body of the response contains information about a request such as state or results.

If you use protobuf for data serialization, the Java and .NET client libraries provide methods and classes to deserialize the protobuf responses.

- The API supports both the synchronous and asynchronous modes of the server.

Note The examples and graphics that follow use JSON as the data serialization format.

RESTful API for MATLAB Function Execution

| Type | Purpose |
|--------------------------------------|--|
| “Synchronous Execution” on page 1-3 | Make synchronous requests to the server |
| “Asynchronous Execution” on page 1-5 | Make asynchronous requests to the server |

RESTful API for Server Discovery and Diagnostics

| Type | Purpose |
|----------------------------------|--|
| “Discovery Service” on page 1-10 | Discover MATLAB functions deployed on the server |
| “Health Check” on page 1-14 | Check server health |

Synchronous Execution

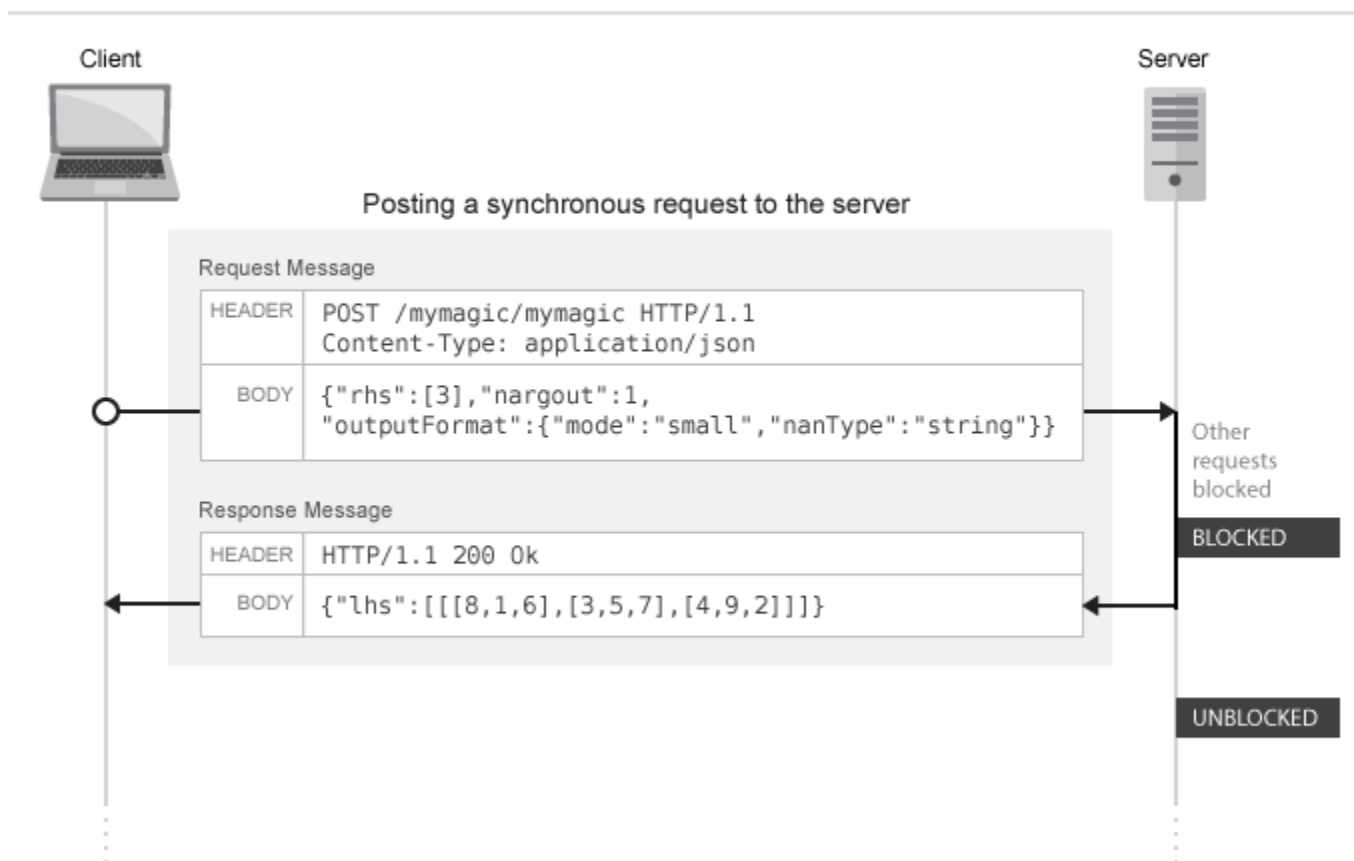
In synchronous mode, after a client posts a request, the server blocks all further requests until it has completed processing the original request. After processing is complete, the server automatically returns a response to the client.

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.

Synchronous



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 25 comma-separated values. These values are the output of the deployed MATLAB function `mymagic`, represented in column-major format. The MATLAB code for the `mymagic` function follows.

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

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

JavaScript Implementation

With the JavaScript implementation of the RESTful API, you include the script within the `<script>` `</script>` tags of an HTML page. When you open this HTML page in a web browser, the server returns the values of the `mymagic` function. Note that the server needs to have CORS enabled for JavaScript code to work. For more information on how to enable CORS, 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 server
              document.getElementById("demo").innerHTML = '<p>' + result.lhs[0].mwdata;
            }
            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 example uses Python 2.x. If you are using Python 3.x, you must 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 “Web-Based Tool Using RESTful API, JSON, and JavaScript” on page 4-2.

Asynchronous Execution

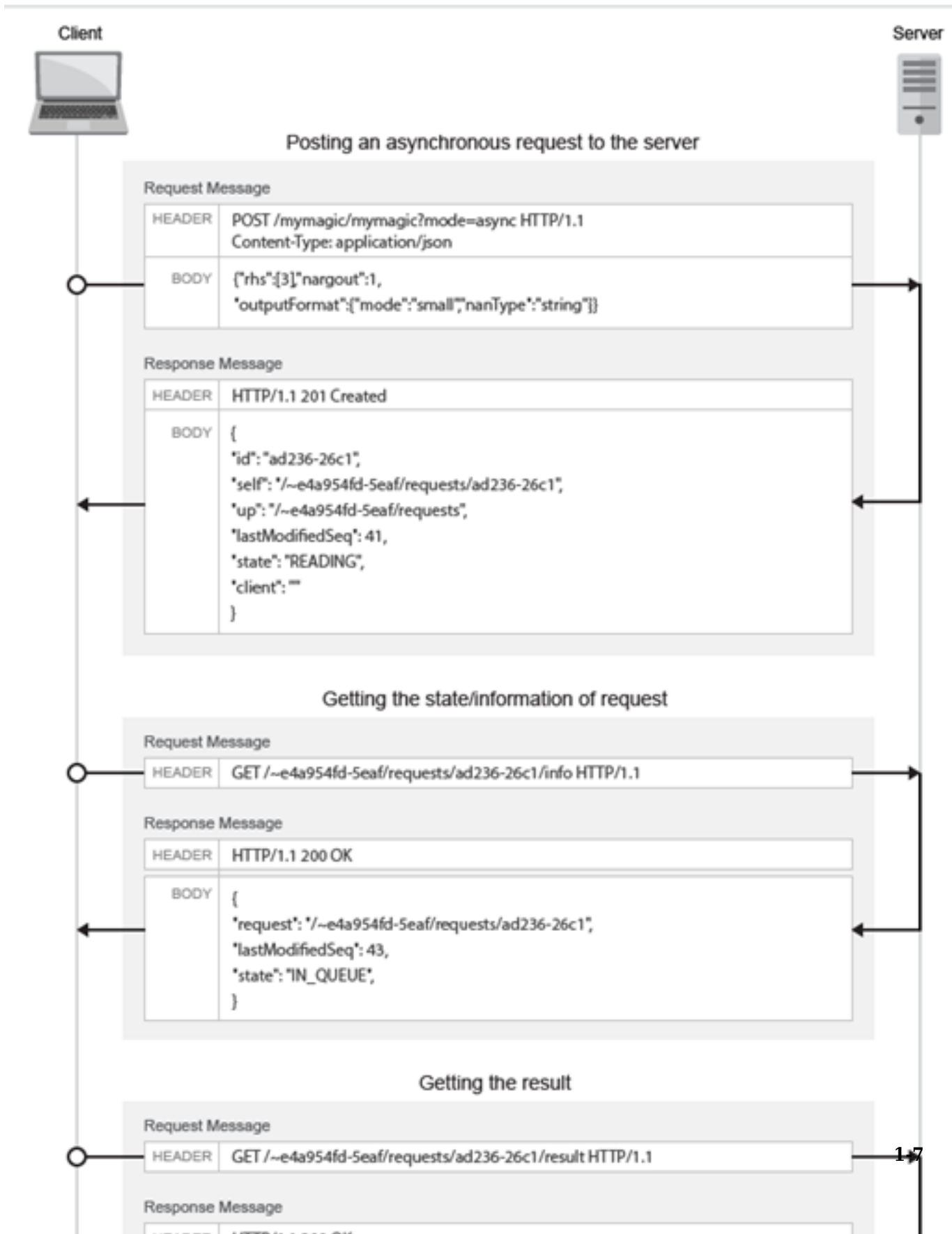
In asynchronous mode, a client is able to post multiple requests, and in each case the server responds by creating a new resource and returning a unique URI corresponding to each request. The URI is encapsulated within the body of the response message. The client can use the URI returned by the server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

RESTful API Calls for Asynchronous Mode

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

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

Asynchronous



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

This example shows how to use the RESTful API and JSON for asynchronous execution using JavaScript. When you execute this example, the server returns a list of 100 comma-separated values. These values are the output of the deployed MATLAB function `mymagic`, represented in column-major format. The MATLAB code for the `mymagic` function follows.

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

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

Code:

`restApiAsyncMagicJavaScript.html`

```
<!DOCTYPE html>
<html>

  <head>
    <title>Magic Square</title>
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js"></script>
    <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=" + clientID;
      // 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 POST method
        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;
              }
            }
          });
        }, 1000);
      }
    </script>
  </head>
</html>
```

```

    }

    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>

```

Asynchronous Request Execution on Azure Using HTTPS

A MATLAB Production Server deployment on Azure® provides an HTTPS endpoint URL to invoke MATLAB functions deployed to the server. The Azure application gateway provides the HTTPS endpoint URL.

The Azure application gateway provides cookie-based session affinity, where it uses cookies to keep a user session on the same server. On receiving a request from a client program, the application gateway sets the `Set-Cookie` HTTP response header with information about the server virtual machine (VM) that processes the request. A client program that uses asynchronous requests to execute a MATLAB function deployed to the server must set the `Cookie` HTTP request header with

the value of the Set -Cookie header for all subsequent requests. This ensures that same server VM that processes the first request processes all subsequent requests for that session.

For more information about the architecture and resources for MATLAB Production Server on Azure, see “Architecture and Resources on Azure” and “Architecture and Resources on Azure”.

Discovery Service

Use the discovery service to learn about the MATLAB functions that you deploy to the server. The discovery service returns information about the 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. Each archive contains information about the deployed MATLAB functions and their function signatures. For more information about the JSON object, see “JSON Response Object” on page 1-11.

To use the discovery service, you must enable the discovery service on the server by setting the `--enable-discovery` property in the `main_config` server configuration file.

To get useful information when using the discovery service, you must include a JSON file containing function signatures of the MATLAB functions that you want to deploy when creating the deployable archive. For information on how to create a deployable archive, see “Create a Deployable Archive for MATLAB Production Server”. For information about creating the JSON file containing function signatures, see “MATLAB Function Signatures in JSON” on page 1-16.

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": {
    "<Name of the CTF archive>":
    {
      "archiveSchemaVersion": "1.1.0",
      "archiveUuid": "<ID of the CTF archive>",
      "name": "<Name of the CTF archive>",
      "matlabRuntimeVersion": "<MATLAB Runtime version number>",
    }
  }
  "typedefs":
  {
    "<struct_name>": {
      "help": "<field_description>",
      "mwtype": "struct",
      "fields": [
        { "name": "<field_name>", "mwtype": [ "<field_matlab_type>" ], "mwsizes": [ "<size1>, ..., <sizeN>" ], "help": "<field_description>" }
      ]
    }
    "<homogeneous_cell_name>":
    {
      "help": "<field_description>",
      "mwtype": "cell",
      "elements": { "name": "<field_name>", "mwtype": [ "<field_matlab_type>" ], "mwsizes": [ "<size1>, ..., <sizeN>" ], "help": "<field_description>" }
    }
    "<heterogeneous_cell_name>":
    {
      "help": "<field_description>",
      "mwtype": "cell",
      "elements": [
        { "name": "<field_name>", "mwtype": [ "<field_matlab_type>" ], "mwsizes": [ "<size1>, ..., <sizeN>" ], "help": "<field_description>" }
        { "name": "<field_name>", "mwtype": [ "<field_matlab_type>" ], "mwsizes": [ "<size1>, ..., <sizeN>" ], "help": "<field_description>" }
      ]
    }
  }
  "functions":
  {
    "MATLAB_function_name1": {
      "signatures": [
        { "help": "<functionName1_description>",
          "inputs": [
            { "name": "<input1_name>", "mwtype": [ "<field_matlab_type>" ], "help": "<field_description>" },
            { "name": "<input2_name>", "mwtype": [ "<matlab_type>", "size=<array_dimensions>" ], "help": "<field_description>" }
          ],
          "outputs": [
            { "name": "<output1_name>", "mwtype": "<matlab_type>", "help": "<field_description>" }
            { "name": "<output2_name>", "mwtype": [ "<matlab_type>", "size=<array_dimensions>" ], "help": "<field_description>" }
          ]
        }
      ]
    }
  }
}

```

JSON Response Object

Object {;}

| Name | Description |
|------------------------|---|
| discoverySchemaVersion | Version number for the discovery schema. <i>Example value: 1.0.0</i> |
| ► archives | List of deployed archives. See “Archives” on page 1-11. |

Archives

archives

► <Name of the CTF archive>

| Name | Description |
|----------------------|---|
| archiveSchemaVersion | Version number of the archive schema. <i>Example value: 1.1.0</i> |
| archiveUuid | Unique identifier. |
| name | Name of the CTF archive. |
| matlabRuntimeVersion | MATLAB Runtime version. <i>Example value: 9.5.0</i> |
| typedefs | If you deploy MATLAB functions that require cell arrays or structs as inputs, you must define them as <code>typedefs</code> . For details, see “Typedefs” on page 1-12. |
| ▶ functions | A function object for each function that you deploy. For details, see “Functions” on page 1-13. |

Typedefs

typedefs

▶<cellName>

| elements | |
|----------|---|
| Name | Description |
| name | Name of cell element. <i>Example: "name": "a"</i> |
| type | Data type of element. <i>Example: "type": "double"</i> |
| size | Size of array. <i>Example: "size": ["2,3"]</i> |
| help | Description of cell element. <i>Example: "help": "Operand a"</i> |
| help | Description for cell. <i>Example: "help": "cell help"</i> |
| type | cell |

▶<struct_name>

| fields | |
|---------------|--|
| Name | Description |
| name | Name of struct field. <i>Example: "name": "my_field_name"</i> |
| type | Data type of field value. <i>Example: "type": "char"</i> |
| size | Size of struct. <i>Example: "size": ["2,3"]</i> |
| help | Description for struct element. <i>Example: "help": "description for my_field_name"</i> |
| help | Description for struct. <i>Example: "help": "struct help"</i> |
| type | struct |

Functions**functions**

► <matlabFunctionName>

↳ signatures

| inputs | |
|---------------|--|
| Name | Description |
| name | Name of input or output parameter. <i>Example: "name": "input1"</i> |
| mwtype | MATLAB data type. <i>Example: "mwtype": "double"</i> |
| mwsiz | Size of data. <i>Example: "mwsiz": ["2,3"]</i> |
| help | Description for input arguments. <i>Example: "help": "input1 description"</i> |

| outputs | |
|---------|---|
| Name | Description |
| name | Name of input or output parameter. <i>Example: "name": "output1"</i> |
| mwtype | MATLAB data type. <i>Example: "mwtype": "double"</i> |
| mwsiz | Size of data. <i>Example: "mwsiz": ["2,3"]</i> |
| help | Description for output parameters. <i>Example: "help": "output1 description"</i> |

Health Check

Use the health check API to determine if the server has a valid license and is able to process HTTP requests. The health check classifies the server as healthy or unhealthy depending on whether the server has a valid license and can communicate with the network license manager. For more information on licensing, see “Manage Licenses for MATLAB Production Server”.

| Server Health | HTTP Response Status Code | Description |
|---------------|---------------------------|--|
| Healthy | 200 OK | <p>A server is healthy when it is in one of the following states:</p> <ul style="list-style-type: none"> • The server is operating with a valid license. The server is communicating with the network license manager, and the required number of license keys are checked out. • The server has lost communication with the network license manager, but the server is still fully operational and will remain operational until the end of the grace period as specified by the license-grace-period property. |

| Server Health | HTTP Response Status Code | Description |
|---------------|---------------------------|--|
| Unhealthy | 503 Health Check Failed | The server has lost communication with the network license manager for a period of time exceeding the grace period. Request processing has been suspended, but the server is actively attempting to reestablish communication with the network license manager. Request processing resumes if the sever is able to reestablish communication with the License Manager. |

If the health check is successful, the response from the server is a JSON object indicating that the server is healthy.

```
{
  "status": "ok"
}
```

A failed health check indicates that the server is unavailable to process HTTP requests but does not provide additional information about the cause of failure in the response body. Server administrators can use `mps - status` to get detailed information about the server status. You must be on the same system as the server to run `mps - status`.

RESTful API Call for Health Check

| Call | Purpose |
|-------------------|--|
| GET Server Health | Get information about the overall health of the server |

See Also

`mps - status`

More About

- “Create a Server”
- “MATLAB Function Signatures in JSON” on page 1-16
- “JSON Representation of MATLAB Data Types” on page 2-2
- “Create a Deployable Archive for MATLAB Production Server”
- “Verify Server Status”

MATLAB Function Signatures in JSON

For a RESTful client to acquire the function signatures of MATLAB functions deployed to MATLAB Production Server using the discovery API, you must 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*.

If your MATLAB functions have `struct` or `cell` data types as inputs or outputs, you can add their descriptions to the JSON file using *typedef objects*.

The JSON file does not support adding descriptions for `datetime` and `enumeration` values, although your MATLAB functions can have these data types as input or outputs.

You can access the JSON object file from the server by using the “Discovery Service” on page 1-10.

Warning The `validateFunctionSignaturesJSON` function does not support validating MATLAB Production Server `<projectName>functionSignatures.json`.

```

// 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"]
//
// To modify function or parameter help text, search for "purpose" and edit
// the values.
//
// JSON-formatted text below this line.
{
  "_schemaVersion": "<major#>.<minor#>.<patch#>",
  "_typedefs": {
    "<struct_name>": {
      "purpose": "<struct_name_description>",
      "type": "struct",
      "fields": [
        { "name": "<field_name>", "type": ["<field_matlab_type>"], "purpose": "<field_description>" },
        { "name": "<field_name>", "type": ["<field_matlab_type>"], "purpose": "<field_description>" }
      ]
    },
    "cell_name": {
      "purpose": "<cell_name_description>",
      "type": "cell",
      "elements": { "type": "element_matlab_type" }
    }
  },
  "functionName1": {
    "inputs": [
      { "name": "<input1_name>", "type": "<matlab_type>", "purpose": "<input1_name_description>" },
      { "name": "<input2_name>", "type": ["<matlab_type>", "size=<array_dimensions>"], "purpose": "<input2_name_description>" }
    ],
    "outputs": [
      { "name": "<output1_name>", "type": "<matlab_type>", "purpose": "<output1_name_description>" },
      { "name": "<output2_name>", "type": ["<matlab_type>", "size=<array_dimensions>"], "purpose": "<output2_name_description>" }
    ],
    "purpose": "<functionName1_description>"
  },
  "functionName2": {
    "inputs": [
      { "name": "<input1_name>", "type": "<matlab_type>", "purpose": "<input1_name_description>" },
      { "name": "<input2_name>", "type": ["<matlab_type>", "size=<array_dimensions>"], "purpose": "<input2_name_description>" }
    ],
    "outputs": [
      { "name": "<output1_name>", "type": "<matlab_type>", "purpose": "<output1_name_description>" },
      { "name": "<output2_name>", "type": ["<matlab_type>", "size=<array_dimensions>"], "purpose": "<output2_name_description>" }
    ],
    "purpose": "<functionName2_description>"
  }
}

```

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

Function Objects

Function objects automatically inherit their name from the name of the MATLAB functions that you add to the project. The purpose line for the function object is inherited from the function description provided in the MATLAB function. 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"], "purpose":"<input 1 description>"},
      {"name":"in2", "type":["logical"], "purpose":"<input 2 description>"}
    ]
  }
}
```

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. You must specify this property and its corresponding value. 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 a double precision number |
| "single" | Must be a single precision number |

| Value | Argument Description |
|-----------|-----------------------------------|
| "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 unsigned integer |
| "int64" | Must be a 64-bit signed integer |
| "uint64" | Must be a 64-bit unsigned integer |
| "logical" | Must be a logical array |
| "char" | Must be a character array |
| "string" | Must be a string array |

For cell and struct, see “Typedef Object” on page 1-19.

The JSON file does not support adding descriptions for datetime and enumeration values.

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

size – Array Dimensions

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

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

purpose – Description for Argument

The purpose property provides a description for the arguments.

Example: { "name": "in", "type": ["double", "size=1,1", "purpose": "Input argument"] }

If you want to use international characters for the purpose argument, enable UTF-8 support for your server machine.

Typedef Object

A typedef object defines cell arrays and structures. Add a typedef object only if values to the argument objects are cells or structures. The JSON file template that the **Production Server Compiler** app generates does not have this object by default.

In the schema, indicate a typedef object by using the name `_typedefs` with its values as the name of one or more cell or structure objects. The type is the same as the argument object.

Example of Using a Homogeneous Cell Array: If a MATLAB function `sortinput` accepts a cell array as input and returns a cell array as output, and each cell in the input consists of a structure, its JSON representation is as follows.

```
{
  "_schemaVersion": "1.1.0",
```

```

    "_typedefs" : {
      "struct_names_scores_of_students": {
        "purpose": "Names and scores of students",
        "type": "struct",
        "fields": [
          {"name": "Name", "type": "char"},
          {"name": "Score", "type": ["double","size=1,1"]}
        ]
      },
      "cell_student_information": {
        "purpose": "Cell representing student information",
        "type": "cell",
        "elements": {
          "type": "struct:struct_names_scores_of_students"
        }
      }
    },
    "sortinput": {
      "inputs": [
        {
          "name": "unsorted_input",
          "type": ["cell:cell_student_information"],
          "purpose": "Unsorted list of students and their scores"
        }
      ],
      "outputs": [
        {
          "name": "sorted_output",
          "type": ["cell:cell_student_information"],
          "purpose": "Sorted list of students with respect to their scores"
        }
      ]
    }
  }
}

```

Example of Using a Heterogeneous Cell Array: If a MATLAB function `organize` accepts a cell array with length 3 containing a character, a square matrix, and a string as input, and returns a vector of doubles as output, its JSON representation is as follows.

```

{
  "_typedefs": {
    "cell_het_mydata": {
      "purpose": "cell containing character, matrix, and string",
      "type": "cell",
      "elements" : [
        { "type": ["char", "size=1,1"], "purpose": "cell element 1 is a character" },
        { "type": ["double", "size=N,N"], "purpose": "cell element 2 is a square matrix" },
        { "type": "char", "purpose": "cell element 3 is a string" }
      ]
    },
    "organize": {
      "inputs": [
        {
          "name": "data",
          "type": ["cell:cell_het_mydata","size=3,1"],
          "purpose": "heterogenous cell array"
        }
      ],

```



```
        "outputs": [  
            {  
                "name": "numerator",  
                "type": "double",  
                "purpose": "result of function"  
            }  
        ]  
    }  
}
```

See Also

More About

- “Discovery Service” on page 1-10
- “JSON Representation of MATLAB Data Types” on page 2-2
- “Create a Deployable Archive for MATLAB Production Server”

JSON Representation of MATLAB Data Types

JSON Representation of MATLAB Data Types

JavaScript Object Notation or JSON is a text-based data interchange format that can be used across programming languages. Since JSON is independent of programming language, you can represent MATLAB data types in JSON. 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 notation formats: *small* and *large*.

- Small notation 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. You can use small notation to represent scalar and multidimensional `double` and `logical` data types, scalar and 1-by-N `char` data type and scalar `struct`.
- Large notation 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 uses large notation by default.*

A JSON object in the large notation contains the following property name-value pairs.

| Property Name | Property Value | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---------------|---|-------------------------------|--|----------|--|--------|--|---------|--|---------|--|----------|--|---------|--|----------|--|---------|--|----------|--|-----------|--|--------|--|----------|--|--------|--|----------|--|------------|--|-------------------------------|--|--|--|
| "mwdata" | JSON array representing the actual data. Specify the property value by enclosing the data as a comma-separated list within <code>[]</code> . | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| "mwsize" | JSON array representing the dimensions of the data. Specify the property value by enclosing the dimensions as a comma-separated list within <code>[]</code> . | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| "mwtype" | JSON string representing the type of data. Specify the property value within <code>" "</code> . <table border="0" style="margin-left: auto; margin-right: auto;"> <tr> <td>"double"</td> <td> </td> <td>"single"</td> <td> </td> <td>"int8"</td> <td> </td> </tr> <tr> <td>"uint8"</td> <td> </td> <td>"int16"</td> <td> </td> <td>"uint16"</td> <td> </td> </tr> <tr> <td>"int32"</td> <td> </td> <td>"uint32"</td> <td> </td> <td>"int64"</td> <td> </td> </tr> <tr> <td>"uint64"</td> <td> </td> <td>"logical"</td> <td> </td> <td>"char"</td> <td> </td> </tr> <tr> <td>"struct"</td> <td> </td> <td>"cell"</td> <td> </td> <td>"string"</td> <td> </td> </tr> <tr> <td>"datetime"</td> <td> </td> <td colspan="4">"<class name of enumeration>"</td> </tr> </table> | "double" | | "single" | | "int8" | | "uint8" | | "int16" | | "uint16" | | "int32" | | "uint32" | | "int64" | | "uint64" | | "logical" | | "char" | | "struct" | | "cell" | | "string" | | "datetime" | | "<class name of enumeration>" | | | |
| "double" | | "single" | | "int8" | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| "uint8" | | "int16" | | "uint16" | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| "int32" | | "uint32" | | "int64" | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| "uint64" | | "logical" | | "char" | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| "struct" | | "cell" | | "string" | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| "datetime" | | "<class name of enumeration>" | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| "mwcomplex" | For complex numbers, set the property value to JSON <code>true</code> . | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

MATLAB Compiler SDK™ provides the following utility functions for data conversion between MATLAB and JSON.

| Function Name | Purpose |
|---|--|
| <code>mps.json.encoderrequest</code> (MATLAB Compiler SDK) | Convert MATLAB data in a server request to JSON text using MATLAB Production Server JSON schema. |
| <code>mps.json.decoderresponse</code> (MATLAB Compiler SDK) | Convert JSON text from a server response to MATLAB data. |
| <code>mps.json.encode</code> (MATLAB Compiler SDK) | Convert MATLAB data to JSON text using MATLAB Production Server JSON schema. |
| <code>mps.json.decode</code> (MATLAB Compiler SDK) | Convert a character vector or string in MATLAB Production Server JSON schema to MATLAB data. |

The RESTful API supports the following MATLAB data types.

Numeric Types: double, single and Integers

- The `mldata` property must be a JSON array of JSON numbers.
- The `mwtpe` property can be any of `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`.
- *You cannot represent scalar or multidimensional single and integer types using JSON small notation.*
- Starting in R2020a, `int64` and `uint64` numbers maintain precision and range in their JSON representation as they are not converted to `double`.

Scalar Numeric Types: double, single and Integers

- The `mldata` property must be a JSON array containing one JSON number representing the MATLAB scalar value.
- The `mwsie` property must be a JSON array containing 1,1.

JSON Representation of Scalar Numeric Types: double, single and Integers

| MATLAB Data Type | JSON Small Notation | JSON Large Notation |
|--|-------------------------|---|
| <code>single</code> , <code>int8</code> , <code>uint8</code> , <code>int16</code> , <code>uint16</code> , <code>int32</code> , <code>uint32</code> , <code>int64</code> , <code>uint64</code> | No small representation | { "mldata": [JSON number], "mwsie": [1,1], "mwtpe": "single" "int8" "uint8" "int32" "uint32" "int64" } |
| <code>double</code> | JSON number | { "mldata": [JSON number], "mwsie": [1,1], "mwtpe": "double" } |

| MATLAB Data: Scalar Numerics | JSON Small Notation | JSON Large Notation |
|------------------------------|-------------------------|--|
| int8(23) | No small representation | { "mwdata": [23], "mwsizes": [1,1], "mwtype": "int8" } |
| uint8(27) | No small representation | { "mwdata": [27], "mwsizes": [1,1], "mwtype": "uint8" } |
| single(20.15) | No small representation | { "mwdata": [20.15], "mwsizes": [1,1], "mwtype": "single" } |
| intmax('int64') | No small representation | { "mwdata": [9223372036854775807], "mwsizes": [1,1], "mwtype": "int64" } |
| double(12.905) | 12.905 | { "mwdata": [12.905], "mwsizes": [1,1], "mwtype": "double" } |
| 42 | 42 | { "mwdata": [42], "mwsizes": [1,1], "mwtype": "double" } |

Multidimensional Numeric Types: double, single and Integers

- The mwdata property must be a JSON array containing data from multidimensional arrays in column-major order. This ordering corresponds to the default memory layout in MATLAB.
- You must represent double arrays, except N-by-1 double arrays, with nested JSON arrays when using small notation
- You cannot represent multidimensional single and integer types using JSON small notation.

| MATLAB Data: Multidimensional double Array | JSON Small Notation | JSON Large Notation |
|--|---------------------|---|
| [1,2,3;... 4,5,6] | [[1,2,3],[4,5,6]] | { "mwdata": [1,4,2,5,3,6], "mwsizes": [2,3], "mwtype": "double" } |

| MATLAB Data: Multidimensional double Array | JSON Small Notation | JSON Large Notation |
|---|--|---|
| [1, NaN, -Inf;... 2, 105, Inf] | [[1,{"mwdata": "NaN"},{"mwdata": "-Inf"}],[2,105,{"mwdata": "Inf"}]] | { "mwdata": [1, 2, "NaN", 105, "-Inf", "mwsizedata": [2,3], "mwtype": "double" } |
| [1 2; 4 5; 7 8] | [[1, 2], [4, 5], [7, 8]] | { "mwdata": [1,4,7,2,5,8], "mwsizedata": [3,2], "mwtype": "double" } |
| 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]]]] | { "mwdata": [1,3,5,2,4,6,7,9,11,8,10,12], "mwsizedata": [3,2,2], "mwtype": "double" } |
| [17;500] | [17,500] | { "mwdata": [17,500], "mwsizedata": [2,1], "mwtype": "double" } |
| [17,500] | [[17,500]] | { "mwdata": [17,500], "mwsizedata": [1,2], "mwtype": "double" } |

Numeric Types: NaN, Inf, and -Inf

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

| MATLAB Data: NaN, Inf, and -Inf | JSON Small Notation | JSON Large Notation |
|---------------------------------|---------------------------------|---|
| NaN | <code>{"mldata": "NaN"}</code> | <pre>{ "mldata": ["NaN"], "mssize": [1,1], "mwtype": "double" }</pre> <p>or</p> <pre>{ "mldata": [{"mldata": "NaN"}], "mssize": [1,1], "mwtype": "double" }</pre> |
| Inf | <code>{"mldata": "Inf"}</code> | <pre>{ "mldata": ["Inf"], "mssize": [1,1], "mwtype": "double" }</pre> <p>or</p> <pre>{ "mldata": [{"mldata": "Inf"}], "mssize": [1,1], "mwtype": "double" }</pre> |
| -Inf | <code>{"mldata": "-Inf"}</code> | <pre>{ "mldata": ["-Inf"], "mssize": [1,1], "mwtype": "double" }</pre> <p>or</p> <pre>{ "mldata": [{"mldata": "-Inf"}], "mssize": [1,1], "mwtype": "double" }</pre> |

Numeric Types: Complex Numbers

- The `mldata` property values must contain the real and imaginary parts of the complex number represented side by side.
- You must set an additional property `mwcomplex` with the value of `true`.
- The `mwtype` property can be any of `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`.
- *You cannot represent complex numbers using small notation.*

JSON Representation of Complex Numbers

| MATLAB Data | JSON Large Notation |
|--|--|
| <code>a + bi</code> | <pre>{ "mwcomplex": true, "mwdata": [a,b], "mwsizes": [1,1], "mwtype": "double" }</pre> |
| MATLAB Data: Scalar Complex Number | JSON Large Notation |
| <code>int32(3 + 4i)</code> | <pre>{ "mwcomplex": true, "mwdata": [3,4], "mwsizes": [1,1], "mwtype": "int32" }</pre> |
| MATLAB Data: Multidimensional Array of Complex Numbers | JSON Large Notation |
| <code>[1 - 2i;... 3 + 7i]</code> | <pre>{ "mwcomplex": true, "mwdata": [1, -2, 3, 7], "mwsizes": [2,1], "mwtype": "double", }</pre> |

Character Array

- The `mwdata` property must be an array of JSON strings.
- The `mwtype` property must have the value of `char`.
- You can represent scalar characters and 1-by-N character arrays in small notation.
- *You cannot represent multidimensional character arrays in large notation.*

JSON Representation of char

| MATLAB Data Type | JSON Small Notation (for scalar and single dimensional character arrays) | JSON Large Notation |
|-------------------|--|--|
| <code>char</code> | JSON string | <pre>{ "mwdata": [JSON string], "mwsizes": [<char dimensions>], "mwtype": "char" }</pre> |

| MATLAB Data: Scalar and Single-dimensional Character Array | JSON Small Notation | JSON Large Notation |
|--|---------------------|--|
| 'a' | "a" | { "mwdata": ["a"], "mwsize": [1,1], "mwtype": "char" } |
| 'hey, jude' | "hey, jude" | { "mwdata": ["hey, jude"], "mwsize": [1,9], "mwtype": "char" } |

| MATLAB Data: Multidimensional Character Array | JSON Large Notation |
|---|---|
| ['boston';... '123456'] | { "mwdata": ["b1o2s3t4o5n6"], "mwsize": [3,4], "mwtype": "char" } |

Logical

- The `mwdata` property must contain only JSON `true` or `false` boolean values. For multidimensional `logical` data, represent the values in column-major order.
- The `mwtype` property must have the value of `logical`.
- In the small notation, you must represent multidimensional `logical` arrays with nested JSON arrays.

JSON Representation of Logical

| MATLAB Data Type | JSON Small Notation | JSON Large Notation |
|----------------------|---------------------------|--|
| <code>logical</code> | <code>true false</code> | { "mwtype": "logical", "mwsize": [1,1], "mwdata": [true false] } |

| MATLAB Data: Scalar logical | JSON Small Notation | JSON Large Notation |
|---|---------------------|---|
| <code>logical(1)</code> or <code>true</code> | <code>true</code> | { "mwdata": [true], "mwsize": [1,1], "mwtype": "logical" } |
| <code>logical(0)</code> or <code>false</code> | <code>false</code> | { "mwdata": [false], "mwsize": [1,1], "mwtype": "logical" } |

| MATLAB Data: Multidimensional logical Array | JSON Small Notation | JSON Large Notation |
|---|------------------------------|---|
| [true,false;... true,false;... true,false] | [[[true,false],[true,false], | [true,false]]] "mwdata": [true,true,true,false,fa "mwsiz e": [3,2], "mwtype": "logical" } |

Cell Array

- The `mwdata` property must be a JSON array that contains the values of the cells in their JSON representation.
- The `mwtype` property must have the value of `cell`.
- *You cannot represent cell arrays using small notation.*

| MATLAB Data Type | JSON Large Notation |
|------------------|---|
| cell | { "mwdata": [<cell data>], "mwsiz e": [<cell dimensions>], "mwtype": "cell" } |

- Although you must represent cell arrays using large notation only, if the data type of a cell element supports small notation, you can represent that element in small notation when encoding the cell array in JSON.

The following table shows an example.

| MATLAB Data: cell Array | JSON Large Notation with some cell elements in Small Notation | JSON Large Notation with all cell elements in Large Notation |
|--|---|--|
| <code>{'Primes',[10 23 199],[false,true,'maybe']}</code> | <pre>{ "mwdata": ["Primes", [[10, 23, 199]], { "mwdata": [false, true, "maybe"], "mwsizes": [1, 3], "mwtype": "cell" }], "mwsizes": [1, 3], "mwtype": "cell" }</pre> | <pre>{ "mwdata": [{ "mwdata": ["Primes"], "mwsizes": [1, 6], "mwtype": "char" }, { "mwdata": [10, 23, 199], "mwsizes": [1, 3], "mwtype": "double" }, { "mwdata": [false, true, "maybe"], "mwsizes": [1, 1], "mwtype": "logical" }, { "mwdata": [true], "mwsizes": [1, 1], "mwtype": "logical" }, { "mwdata": ["maybe"], "mwsizes": [1, 5], "mwtype": "char" }], "mwsizes": [1, 3], "mwtype": "cell" }, { "mwdata": [10, 23, 199], "mwsizes": [1, 3], "mwtype": "double" }, { "mwdata": [false, true, "maybe"], "mwsizes": [1, 3], "mwtype": "cell" }], "mwsizes": [1, 3], "mwtype": "cell" }</pre> |

- For more information on MATLAB cell data type, see `cell`.

Structure Array

- The `mwdata` property must be a JSON object that contains name-value pairs, where the name matches a *field* in the `struct` and value is a JSON array that represents the data in the field.
- The `mwtype` property must have the value of `struct`.
- Although you must represent multidimensional `struct` arrays using JSON large notation, if the data type of a `struct` value supports small notation, you can represent that value in small notation when encoding the `struct` in JSON.
- *You can represent only a scalar struct in small notation.*

JSON Representation of struct arrays

| MATLAB Data Type | JSON Small Notation (valid only for a scalar struct) | JSON Large Notation |
|---------------------|--|---|
| <code>struct</code> | JSON object | <pre>{ "mwdata": {<struct data>} "mwsizes": [<struct dimensions>], "mwtype": "struct" }</pre> |

-

| MATLAB Data: Scalar Structure Array | JSON Small Notation | JSON Large Notation |
|--|--|---|
| <pre>struct('name','John Smith', 'age',15)</pre> | <pre>{'age':15, 'name':'John Smith'}</pre> | <pre>{ "age": 15, "mwdtype": "double", "name": "John Smith", "mwsz": [1, 10], "mwtpe": "char" }</pre> |

- Although you can represent 1-by-1 struct arrays in small notation, if the data type of a struct value does not support small notation, you must represent that value in large notation when encoding the struct in JSON.

| MATLAB Data: 1-by-1 Structure Array | JSON Small Notation with some struct values in Large Notation | JSON Large Notation with all struct values in Large Notation |
|---|---|---|
| <pre>struct('Name', {'Casper', 'Ghost'}, ... 'Age', {[14,17,18]}, ... 'Date', {736676})</pre> | <pre>{ "Age": [14, 17, 18], "Date": 736676, "Name": ["Casper", "Ghost"] }</pre> | <pre>{ "Age": [14, 17, 18], "Date": 736676, "Name": ["Casper", "Ghost"] }</pre> |

- Although you must represent multidimensional struct arrays using JSON large notation, if the data type of a struct value supports small notation, you can represent that value in small notation when encoding the struct in JSON.

| MATLAB Data: Multidimensional Structure Array | JSON Large Notation with some struct values in Small Notation | JSON Large Notation with all struct values in Large Notation |
|--|--|---|
| <pre>struct('Name', {'Casper', 'Ghost';... 'Genie', 'Wolf'},... 'Ages', {14,17;... 20,23})</pre> | <pre>{ "mwdata":{ "Ages": [14,20,17,23], "Name": ["Casper", "Ghost", "Genie", "Wolf"] }, "mwsizedata": [2,2], "mwtype": "struct" }</pre> | <pre>{ "mwdata":{ "Ages": [{ "mwdata": [14], "mwsizedata": [1,1], "mwtype": "double" }, { "mwdata": [20], "mwsizedata": [1,1], "mwtype": "double" }, { "mwdata": [17], "mwsizedata": [1,1], "mwtype": "double" }, { "mwdata": [23], "mwsizedata": [1,1], "mwtype": "double" }], "Name": [{ "mwdata": ["Casper"], "mwsizedata": [1,6], "mwtype": "char" }, { "mwdata": ["Genie"], "mwsizedata": [1,5], "mwtype": "char" }, { "mwdata": ["Ghost"], "mwsizedata": [1,5], "mwtype": "char" }, { "mwdata": ["Wolf"], "mwsizedata": [1,4], "mwtype": "char" }] }, "mwsizedata": [2,2], "mwtype": "struct" }</pre> |

- For more information on MATLAB struct data type, see `struct`.

String Array

- The `mwdata` property must be a JSON array containing strings in column-major order.
- The `mwtype` property must have the value of `string`.
- *You cannot represent string arrays using small JSON notation.*

JSON Representation of string arrays

| MATLAB Data Type | JSON Large Notation |
|--|---|
| string | { "mwdata": [JSON string], "mwsizes": [<string dimensions>], "mwtype": "string" } |
| MATLAB Data: Scalar, Single-dimensional, Multidimensional, and missing (MATLAB) string Arrays | JSON Large Notation |
| "abc" | { "mwdata": ["abc"], "mwsizes": [1, 1], "mwtype": "string" } |
| ["abc"] | { "mwdata": ["abc"], "mwsizes": [1, 1], "mwtype": "string" } |
| ["abc" "de"] | { "mwdata": ["abc", "de"], "mwsizes": [1, 2], "mwtype": "string" } |
| ["abc" "de"; "fg" "hi"] | { "mwdata": ["abc", "fg", "de", "hi"], "mwsizes": [2, 2], "mwtype": "string" } |
| string(missing) | { "mwdata": [{"mwdata": "missing"}], "mwsizes": [1, 1], "mwtype": "string" } |

- For more information on MATLAB string data type, see `string`.

Enumeration

- The `mwdata` property must be a JSON array of strings denoting the enumeration members.
- The `mwtype` property must be set to the class of the enumerations in the array.
- *You cannot represent an enumeration using small JSON notation.*

JSON Representation of enumeration

| MATLAB Data Type | JSON Large Notation |
|------------------|---|
| enumeration | { "mwdata": [JSON string], "mwsiz e": [<enumeration dimensions>], "mwtype": "<class name of enumeration>" } |

The following table shows examples of JSON representation of an enumeration.

Use the following enumeration for the examples. For more information, see “Define Enumeration Classes” (MATLAB).

```
classdef Colors
    enumeration
        Black Blue Red
    end
end
```

| MATLAB Data: Object of Enumeration Class | JSON Large Notation |
|--|--|
| b = Colors.Black | { "mwdata": ["Black"], "mwsiz e": [1, 1], "mwtype": "Colors" } |
| b = [Colors.Black Colors.Blue] | { "mwdata": ["Black", "Blue"], "mwsiz e": [1, 2], "mwtype": "Colors" } |

- For more information on MATLAB enumeration data type, see `enumeration`.

Datetime Array

- The `mwdata` property must be a JSON object containing name-value pairs for `TimeStamp` and optionally for `LowOrderTimeStamp`. Values for `TimeStamp` and `LowOrderTimeStamp` are JSON representation of the `double` data type.
 - The `TimeStamp` property values represent the POSIX time in milliseconds elapsed since 00:00:00 1-Jan-1970 UTC (Coordinated Universal Time).
 - The `LowOrderTimeStamp` property values represent additional resolution in the timestamp. Use this property to maintain precision past milliseconds.
 - Although you must represent `datetime` arrays using large notation only, since `TimeStamp` and `LowOrderTimeStamp` represent values of the `double` data type which supports small notation, you can represent `TimeStamp` and `LowOrderTimeStamp` using small notation when encoding `datetime` arrays in JSON.
- The `mwmetadata` property must be a JSON object containing name-value pairs for `TimeZone` and `Format`. Values for `TimeZone` and `Format` are JSON representation of the `char` data type.
 - The values of the `TimeZone` and `Format` properties contain metadata necessary for recreating the `datetime` values with timezones in MATLAB in their original display format. This

metadata is necessary because the numeric values contained in the `TimeStamp` and `LowOrderTimeStamp` arrays are calculated with respect to UTC.

- You can specify `TimeZone` and `Format` properties for `NaN` and `Inf` `datetime` array values.
- Although you must represent `datetime` arrays using large notation only, since `TimeZone` and `Format` represent values of the `char` data type which supports small notation, you can represent `TimeZone` and `Format` using small notation when encoding `datetime` arrays in JSON.
- The value for `TimeZone` can be empty.
- The default value for `Format` depends on your system locale. For more information, see “Default `datetime` Format” (MATLAB).
- The `mwtype` property must have the value of `datetime`.
- *You cannot represent `datetime` arrays using small JSON notation.*

JSON Representation of `datetime` arrays

| MATLAB Data Type | JSON Large Notation |
|-----------------------|--|
| <code>datetime</code> | <pre>{ "mwdata": { "LowOrderTimeStamp": <JSON number> "TimeStamp": <JSON number> }, "mwmetadata": { "TimeZone": <JSON string>, "Format": <JSON string> }, "mwsizes": [<datetime array dimensions>], "mwtype": "datetime" }</pre> |

| MATLAB Data: Scalar datetime Array | JSON Large Notation with mwdata and mwmetadata in Small Notation | JSON Large Notation with mwdata and mwmetadata in Large Notation |
|------------------------------------|---|---|
| <pre>datetime(2015, 3, 24);</pre> | <pre>{ "mwdata": { "TimeStamp": 1.4271552E+12 }, "mwmetadata": { "Format": "dd-MMM-uuuu", "TimeZone": "" }, "mwsizes": [1, 1], "mwtype": "datetime" }</pre> | <pre>{ "mwdata": { "mwdata": { "TimeStamp": { "mwdata": [1.4271552E+12] }, "mwsizes": [1, 1], "mwtype": "double" } }, "mwmetadata": { "Format": { "mwdata": ["dd-MMM-uuuu"] }, "mwsizes": [1, 11], "mwtype": "char" }, "TimeZone": { "mwdata": [""], "mwsizes": [0, 0], "mwtype": "char" } }, "mwsizes": [1, 1], "mwtype": "datetime" }</pre> |

The following table shows JSON representation for a `datetime` row vector. Since `LowOrderTimeStamp` and `TimeStamp` contain `double` values, you need to use nested JSON arrays when representing multidimensional (except N-by-1) arrays of `LowOrderTimeStamp` and `TimeStamp` in small notation.

| MATLAB Data: datetime Row Vector | JSON Large Notation with mwdata and mwmetaddata in Small Notation | JSON Large Notation with mwdata and mwmetaddata in Large Notation |
|---|---|---|
| <pre>datetime(2018,1,8,10,... 11,12,(1:5)+(1:5)*1e-6,... 'TimeZone','local');</pre> | <pre>{ "mwdata": { "LowOrderTimeStamp": [[9.999999917335E-7, 1.999999998354667E-6, 2.999999999752447E-6, 3.999999996709334E-6, 4.999999998107114E-6]], "TimeStamp": [[1.515424272001E+12, 4.999999998107114E-6, 1.515424272002E+12, 1.515424272003E+12, 1.515424272004E+12, 1.515424272005E+12]], }, "mwmetaddata": { "Format": "dd-MMM-uuu HH:mm:ss", "TimeZone": "America/New_York", }, "mwsizes": [1, 5], "mwtype": "datetime" }</pre> | <pre>{ "mwdata": { "LowOrderTimeStamp": { "mwdata": [9.999999917335E-7, 1.999999998354667E-6, 2.999999999752447E-6, 3.999999996709334E-6, 4.999999998107114E-6], "mwsizes": [1, 5], "mwtype": "double" }, "TimeStamp": { "mwdata": [1.515424272001E+12, 1.515424272002E+12, 1.515424272003E+12, 1.515424272004E+12, 1.515424272005E+12], "mwsizes": [1, 5], "mwtype": "double" } }, "mwmetaddata": { "Format": { "mwdata": ["dd-MMM-uuu HH:mm:ss"], "mwsizes": [1, 20], "mwtype": "char" }, "TimeZone": { "mwdata": ["America/New_York"], "mwsizes": [1, 16], "mwtype": "char" } }, "mwsizes": [1, 5], "mwtype": "datetime" }</pre> |

| MATLAB Data: datetime Column Vector | JSON Large Notation with mwdata and mwmetadata in Small Notation | JSON Large Notation with mwdata and mwmetadata in Large Notation |
|---|--|---|
| <pre>datetime(2018,1,8,10,... 11,12,(1:5)+(1:5)*1e-6,... 'TimeZone','local');</pre> | <pre>{ "mwdata": { "LowOrderTimeStamp": [9.99999999177336E-7, 1.999999998354667E-6, 2.999999999752447E-6, 3.999999996709334E-6, 4.999999998107114E-6], "TimeStamp": [1.515424272001E+12, 1.515424272002E+12, 1.515424272003E+12, 1.515424272004E+12, 1.515424272005E+12], }, "mwmetadata": { "Format": "dd-MMM-uuu HH:mm:ss", "TimeZone": "America/New_York", }, "mwsizes": [5, 1], "mwtype": "datetime" }</pre> | <pre>{ "mwdata": { "LowOrderTimeStamp": { "mwdata": [9.99999999177336E-7, 1.999999998354667E-6, 2.999999999752447E-6, 3.999999996709334E-6, 4.999999998107114E-6], "TimeStamp": [1.515424272001E+12, 1.515424272002E+12, 1.515424272003E+12, 1.515424272004E+12, 1.515424272005E+12], "mwsizes": [1, 5], "mwtype": "double" }, "TimeStamp": { "mwdata": [1.515424272001E+12, 1.515424272002E+12, 1.515424272003E+12, 1.515424272004E+12, 1.515424272005E+12], "mwsizes": [5, 1], "mwtype": "double" } }, "mwmetadata": { "Format": { "mwdata": ["dd-MMM-uuu HH:mm:ss"], "mwsizes": [1, 20], "mwtype": "char" }, "TimeZone": { "mwdata": ["America/New_York"], "mwsizes": [1, 16], "mwtype": "char" } }, "mwsizes": [1, 5], "mwtype": "datetime" }</pre> |

| MATLAB Data: NaT and Inf datetime Array | JSON Large Notation with mwdata and mwmetadate in Small Notation | JSON Large Notation with mwdata and mwmetadate in Large Notation |
|---|--|--|
| NaT | <pre>{ "mwdata": { "TimeStamp": { "mwdata": "NaN" } }, "mwmetadate": { "Format": "dd-MMM-uuuu HH:mm:ss", "TimeZone": "" }, "mwsizes": [1, 1], "mwtype": "datetime" }</pre> | <pre>{ "mwdata": { "TimeStamp": { "mwdata": ["NaN"], "mwsizes": [1, 1], "mwtype": "double" } }, "mwmetadate": { "Format": { "mwdata": ["dd-MMM-uuuu HH:mm:ss"], "mwsizes": [1, 20], "mwtype": "char" } }, "TimeZone": { "mwdata": [""], "mwsizes": [0, 0], "mwtype": "char" } }, "mwsizes": [1, 1], "mwtype": "datetime" }</pre> |
| datetime(inf,inf,inf) | <pre>{ "mwdata": { "TimeStamp": { "mwdata": "Inf" } }, "mwmetadate": { "Format": "dd-MMM-uuuu HH:mm:ss", "TimeZone": "" }, "mwsizes": [1, 1], "mwtype": "datetime" }</pre> | <pre>{ "mwdata": { "TimeStamp": { "mwdata": ["Inf"], "mwsizes": [1, 1], "mwtype": "double" } }, "mwmetadate": { "Format": { "mwdata": ["dd-MMM-uuuu HH:mm:ss"], "mwsizes": [1, 20], "mwtype": "char" } }, "TimeZone": { "mwdata": [""], "mwsizes": [0, 0], "mwtype": "char" } }, "mwsizes": [1, 1], "mwtype": "datetime" }</pre> |

- For more information on MATLAB datetime data type, see `datetime`.

Empty Array: []

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

| MATLAB Data: Empty Array | JSON Small Notation | JSON Large Notation |
|--------------------------|---------------------|--|
| [] | [] | <pre>{ "mwdata": [], "mwsizes": [0,0], "mwtypes": "double" "single" "int8" "uint8" "i "int32" "uint32" "i "logical" "char" "c "<class name of enumeration</pre> |

See Also

More About

- “RESTful API” on page 1-2
- “Fundamental MATLAB Classes” (MATLAB)

External Websites

- JSON RFC

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 3-3 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. For information about the data types that the MATLAB Production Server supports, see “JSON Representation of MATLAB Data Types” on page 2-2. |

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-Allow-Origin'

For information on how to enable CORS, see [cors-allowed-origins](#).

See Also

More About

- “JSON Representation of MATLAB Data Types” on page 2-2
- “RESTful API” on page 1-2

Examples: RESTful API and JSON

Web-Based Tool Using RESTful API, JSON, and 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:

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

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 4-2

“Step 3: Place the Deployable Archive on a Server” on page 4-3

“Step 4: Enable Cross-Origin Resource Sharing (CORS) on the Server” on page 4-3

“Step 5: Write JavaScript Code using the RESTful API and JSON” on page 4-3

“Step 6: Embed JavaScript within HTML Code” on page 4-4

“Step 7: Run Example” on page 4-6

Step 1: Write MATLAB Code

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

```
function price = pricercalc(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 `pricercalc.m`.
- 4 Under **Archive information**, change `pricercalc` 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 “Supported MATLAB Runtime Versions” for more information.
- 2 Create a server using `mps -new`. See “Create a Server” for more information. If you have not 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 domain 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

Write the following JavaScript code using the RESTful API on page 1-2 and JSON Representation of MATLAB Data Types on page 2-2 as a guide. 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 Price: "
      else { document.getElementById("error").innerHTML = "Error: " + result.error
    }
  }
  else { document.getElementById("error").innerHTML = "Error:" + request.statusText
        document.getElementById("response").innerHTML = "Status: " + request.status + "<br>"
        + "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 field
//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 formula.<p>
  You run this example by entering the following known values into a simple graphical interface
  <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>
  <h3>M: Face Value </h3>
  <input id="facevalue_value" type="number" maxlength="4" oninput="updatePrice('facevalue_value', 'facevalue_value')" />
  <input type="range" id="facevalue_slider" value="0" min="0" max="10000" onchange="updatePrice('facevalue_value', 'facevalue_value')"/>
  <h3>C: Coupon Payment </h3>
  <input id="coupon_payment_value" type="number" maxlength="4" oninput="updatePrice('coupon_payment_value', 'coupon_payment_value')" />
  <input type="range" id="coupon_payment_slider" value="0" min="0" max="1000" onchange="updatePrice('coupon_payment_value', 'coupon_payment_value')"/>
  <h3>N: Number of payments </h3>
  <input id="num_payments_value" type="number" maxlength="4" oninput="updatePrice('num_payments_value', 'num_payments_value')" />
  <input type="range" id="num_payments_slider" value="0" min="0" max="1000" onchange="updatePrice('num_payments_value', 'num_payments_value')"/>
  <h3>i: Interest rate </h3>
  <input id="interest_rate_value" type="number" maxlength="4" step="0.01" oninput="updatePrice('interest_rate_value', 'interest_rate_value')" />
  <input type="range" id="interest_rate_slider" value="0" min="0" max="1" step="0.01" onchange="updatePrice('interest_rate_value', 'interest_rate_value')"/>
  <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

Confirm that the server with the deployed MATLAB function is 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.

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

S: 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

Status: 200
 Status message: OK
 Response text: `{"lhs":[{"mwdata":1079.8542007415617,"mwsize":1,"mwtype":"double"}]}`

See Also

More About

- “Troubleshooting RESTful API Errors” on page 3-2

RESTful APIs

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. The server automatically returns a response once processing is complete. No other HTTP methods are necessary to retrieve the response from the server.

The following sections use JSON as the data serialization format. For an example that shows how to use protobuf as the data serialization format with the Java client API, see “Synchronous RESTful Requests Using Protocol Buffers in the Java Client”, and with the .NET client API, see “Synchronous RESTful Requests Using Protocol Buffers in .NET Client”.

Request

HTTP Method

POST

URI

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

Query Parameters

None.

Content-Type

- `application/json`

Body

| Name | Description | Value-Type |
|----------------------|--|--------------------------------------|
| <code>nargout</code> | 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 <code>nargout</code> . | number |
| <code>rhs</code> | Input arguments to the deployed MATLAB function, specified as an array of comma-separated values. | <code>[arg1, arg2, arg3, ...]</code> |

| Name | Description | Value-Type |
|--------------|---|---|
| outputFormat | <p>Specify whether the MATLAB output in the response should be returned using large or small JSON notation, and whether NaN and Inf should be represented as a JSON string or object. If the mode is not specified or the returned MATLAB data type does not support JSON small notation, the response is represented using JSON large notation.</p> <p>For more information on the JSON representation of MATLAB data types, see “JSON Representation of MATLAB Data Types” on page 2-2.</p> | <pre>{ "mode" : "small large", "nanInfFormat" : "string object" }</pre> |

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

HTTP Status Code

200 OK

Body

| 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**HTTP Status Code**

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

“JSON Representation of MATLAB Data Types” on page 2-2

“Synchronous Execution” on page 1-3

“Example: Synchronous Execution of Magic Square Using RESTful API and JSON” on page 1-3

“Synchronous RESTful Requests Using Protocol Buffers in .NET Client”

“Synchronous RESTful Requests Using Protocol Buffers in the Java Client”

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.

The following sections use JSON as the data serialization format. For an example that shows how to use protobuf as the data serialization format with the Java client API and the .NET client API, see “Asynchronous RESTful Requests Using Protocol Buffers in the Java Client” and “Asynchronous RESTful Requests Using Protocol Buffers in .NET Client”.

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. You can use nargout to specify a subset of these potential outputs. | 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 notation, and whether NaN and Inf should be represented as a JSON string or object. If the mode is not specified or the returned MATLAB data type does not support JSON small notation, the response is represented using JSON large notation. For more information on the JSON representation of MATLAB data types, see "JSON Representation of MATLAB Data Types" on page 2-2. | { "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

HTTP Status Code

201 Created

Body

| 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 <code>self</code> 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-b405-329cb8c373d6",
  "up": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests",
  "lastModifiedSeq": 6,
  "state": "READING",
  "client": ""
}
```

Error

HTTP Status Code

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/ad2363f3-26c1-4d48-88f8-6b7fb615f254
  X-MPS-Start-Time: 003472d705bd1cd2
  Content-Length: 248
Body:
{
  "id": "ad2363f3-26c1-4d48-88f8-6b7fb615f254",
  "self": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/ad2363f3-26c1-4d48-88f8-6b7fb615f254",
  "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 1-5

“Example: Asynchronous Execution of Magic Square Using RESTful API and JSON” on page 1-8

“Web-Based Tool Using RESTful API, JSON, and JavaScript” on page 4-2

“Asynchronous RESTful Requests Using Protocol Buffers in .NET Client”

“Asynchronous RESTful Requests Using Protocol Buffers in the Java Client”
“JSON Representation of MATLAB Data Types” on page 2-2

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.

The following sections use JSON as the data serialization format.

Request

HTTP Method

GET

URI

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

Response

Success

HTTP Status Code

200 OK

Body

| Name | Description | Value-Type |
|------------------------------|--|--|
| <code>id</code> | ID of a particular request. | {id-string} |
| <code>self</code> | URI of particular request. | {request-uri-string} |
| <code>up</code> | URI of a collection of requests tied to a particular client. | {request-collection-uri-string} |
| <code>lastModifiedSeq</code> | Number indicating when a request represented by <code>self</code> was last modified. | {server-state-number} |
| <code>state</code> | State of a request. | {request-state-string} Possible states: READING IN_QUEUE PROCESSING READY ERROR CANCELLED |

| Name | Description | Value-Type |
|--------|---|--------------------|
| 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-18c351abeb5b",
  "up": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests",
  "lastModifiedSeq": 30,
  "state": "READING",
  "client": "786"
}
```

Error

HTTP Status Code

400 NoMatchForQueryParams

404 ResourceNotFound

Sample Call

HTTP

Request:

```
GET /~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/f90c2ff8-4d27-4795-806d-18c351abeb5b HTTP/1.1
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-18c351abeb5b",
  "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/f90c2ff8-4");
xhr.send(data);
```

See Also

[GET State Information](#) | [GET Result of Request](#)

Topics

“Asynchronous RESTful Requests Using Protocol Buffers in the Java Client”

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.

The following sections use JSON as the data serialization format. For an example that shows how to use protobuf as the data serialization format with the Java client API, see “View the Collection of Requests Owned by a Particular Client”.

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

HTTP Status Code

200 OK

Body

| 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. | <pre> "data": [{ "id": {id-string}, "self": [request-uri-string], "up": {request-collection-uri-string}, "lastModifiedSeq": {server-state-number}, "state": {request-state-string}, "client": {client-id-string} }, { "id": {id-string}, "self": {request-uri-string}, "up": {request-collection-uri-string}, "lastModifiedSeq": {server-state-number}, "state": {request-state-string}, "client": {client-id-string} }, ...] </pre> |

Example:

```

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

```

Error**HTTP Status Code**

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-d8470e6e082d",
      "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-a855-6d64192842d8",
      "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/requests?since=15");
xhr.send(data);
```

See Also

GET State Information | GET Representation of Asynchronous Request

Topics

“Asynchronous RESTful Requests Using Protocol Buffers in the Java Client”

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`.

The following sections use JSON as the data serialization format. For an example that shows how to use protobuf as the data serialization format with the Java client API, see “Get the State Information of the Request”.

Request

HTTP Method

GET

URI

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

Response

Success

HTTP Status Code

200 OK

Body

| Name | Description | Value-Type |
|------------------------------|---|---|
| <code>request</code> | URI to current request. | <code>{request-uri-string}</code> |
| <code>lastModifiedSeq</code> | Number indicating when the current request was last modified. | <code>{server-state-number}</code> |
| <code>state</code> | State of current request. | <code>{request-state-string}</code> Possible states: <code>READING</code> <code>IN_QUEUE</code> <code>PROCESSING</code> <code>READY</code> <code>ERROR</code> <code>CANCELLED</code> |

Example:

```
{
  "request": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/c5666088-b087-4bae-aa7d-d8470e6e082d/info",
  "lastModifiedSeq": 19,
  "state": "READY"
}
```

Error**HTTP Status Code**

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-d8470e6e082d/info HTTP/1.1
Host: localhost
Port: 9910
```

Response:

```
Status Code: 200 OK
{
  "request": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/c5666088-b087-4bae-aa7d-d8470e6e082d/info",
  "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/c5666088-b087-4bae-aa7d-d8470e6e082d/info");
xhr.send(data);
```

See Also

[GET Representation of Asynchronous Request](#) | [GET Result of Request](#)

Topics

“Asynchronous RESTful Requests Using Protocol Buffers in the Java Client”

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.

The following sections use JSON as the data serialization format. For an example that shows how to use protobuf as the data serialization format with the Java client API, see “Retrieve the Results of a Request”.

Request

HTTP Method

GET

URI

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

Response

Success

HTTP Status Code

200 OK

Body

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

HTTP Status Code

404 RequestNotFound

410 RequestAlreadyCompleted

410 RequestAlreadyCancelled

410 RequestAlreadyDeleted

500 InternalServerError

Sample Call

HTTP

Request:

```
GET /~f76280c5-b94c-4cd9-8eb6-841532788583/requests/ad063314-ebda-4310-b356-59420058c17c/result
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/ad063314-e");
xhr.send(data);
```

See Also

[GET State Information | DELETE Request](#)

Topics

“Asynchronous RESTful Requests Using Protocol Buffers in the Java Client”

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

HTTP Status Code

204 No Content

Error

HTTP Status Code

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-af8a8905b1fe/cancel
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/ef90fca4-");
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

HTTP Status Code

204 No Content

Error

HTTP Status Code

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-b3f8-6e1e025f9c9b HTTP/1.1
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/requests/31577b5");
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 that you deploy to the server. You receive information about

- all deployed archives with discovery information.
- names of the MATLAB functions that each archive contains.
- names and MATLAB data types of the inputs and outputs for each of the MATLAB functions.
- additional metadata.

If you build a deployable archive (CTF file) without including discovery information, it is not discoverable.

In order to use the discovery service, you must enable the discovery service on the server. Do this by uncommenting the option `--enable-discovery` in the `main_config` server configuration file.

Request

HTTP Method

GET

URI

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

Response

Success

HTTP Status Code

200 OK

Body

For a description of the body, see “JSON Response Object” on page 1-11.

Error

403 DiscoveryDisabled

Sample Call

HTTP

Request:

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

Response:

```
{
  "discoverySchemaVersion": "1.0.0",
  "archives": {
    "mymagic": {
      "archiveSchemaVersion": "1.1.0",
      "archiveUuid": "mymagic_73BCCE8B5FFFB984888169285CBA8A31",
      "name": "mymagic"
      "matlabRuntimeVersion": "9.5.0"

      "functions": {
        "mymagic": {
          "signatures": [
            {
              "help": "Generate a magic square",
              "inputs": [
                {
                  "name": "in",
                  "mwtype": "double",
                  "mwsiz": [],
                  "help": "Dimension of magic square matrix"
                }
              ],
              "outputs": [
                {
                  "name": "out",
                  "mwtype": "double",
                  "mwsiz": [],
                  "help": "Magic square matrix"
                }
              ]
            }
          ]
        }
      }
    }
  }
}
```

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 1-10

“MATLAB Function Signatures in JSON” on page 1-16

Introduced in R2018a

GET Server Health

Get information about the overall health of the server

Description

Use the GET method to determine whether the server is healthy and able to process HTTP requests.

The server is healthy if it has a valid license or has lost communication with the network license manager but is still within the grace period specified by the `license-grace-period` property.

Request

HTTP Method

GET

URI

`http://host:port/api/health`

Response

Success

HTTP Status Code

200 OK

Body

| Name | Description | Value-Type |
|--------|-------------------|------------|
| status | Status of server. | ok |

Example:

```
{
  "status": "ok"
}
```

Error

HTTP Status Code

503 Health check failed

Sample Call

HTTP

Request:

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

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
  "status": "ok"
}
```

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/health");
xhr.send(data);
```

See Also

Topics

“Health Check” on page 1-14

Introduced in R2019b