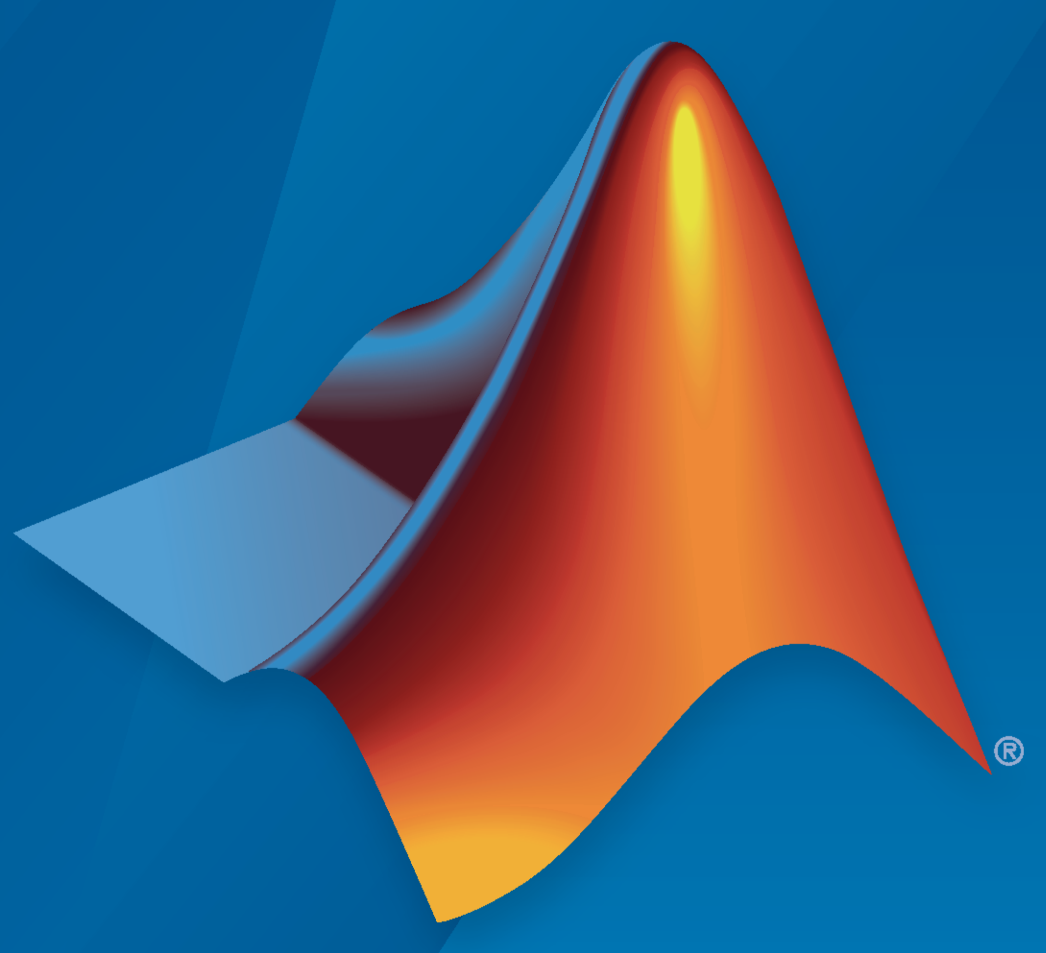
MATLAB® Production Server™

RESTful API and JSON



MATLAB®

R2022b

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 2016–2022 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)
March 2021	Online only	Revised for Version 4.5 (Release R2021a)
September 2021	Online only	Revised for Version 4.6 (Release R2021b)
March 2022	Online only	Revised for Version 5.0 (Release R2022a)
September 2022	Online only	Revised for Version 5.1 (Release R2022b)

Contents

1 Client Programming

RESTful API for MATLAB Function Execution **1-2**
Characteristics of 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 **1-7**
Manage HTTP Cookie **1-8**

RESTful API for Discovery and Diagnostics **1-10**
Characteristics of RESTful API **1-10**
Discovery Service **1-10**
Health Check **1-14**
Metrics Service **1-15**

MATLAB Function Signatures in JSON **1-18**
Function Objects **1-19**
Signature Objects **1-20**
Argument Objects **1-20**
Typedef Object **1-21**

2 JSON Representation of MATLAB Data Types

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**

3 Troubleshooting RESTful API Errors

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

4 Examples: RESTful API and JSON

Create 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**

Create Custom Prometheus Metrics **4-9**
Write MATLAB Code to Create Custom Metrics **4-9**
Deploy MATLAB Function to Server **4-9**
Enable Metrics on Server **4-9**
Execute Deployed Function **4-9**
Query Metrics Service to Retrieve Custom Metrics **4-10**

5 RESTful APIs

Client Programming

RESTful API for MATLAB Function Execution

The MATLAB Production Server RESTful API for MATLAB function execution enables you to evaluate MATLAB functions on remote servers using JSON representation of MATLAB data types and protocol buffers. Protocol buffer support is available only in the Java® and .NET client APIs.

You can write client code that uses the MATLAB Production Server RESTful API in web-based languages such as JavaScript® and embed it in HTML pages. You can then use these web pages to send requests and retrieve responses from a MATLAB Production Server instance. While web-based applications may be more amenable to client code written in JavaScript, you can use any HTTP supported programming language such Java, Python, C++, .NET, and many others to develop client applications.

If client programs make requests from different domains, programmers using JavaScript must verify whether Cross-Origin Resource Sharing (CORS) is enabled on the server. To enable CORS on the server, the server administrator must set the appropriate value for the cors-allowed-origins property in the `main_config` server configuration file.

Characteristics of RESTful API

The RESTful API for MATLAB function execution uses the HTTP request-response model 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. For more information, see “JSON Representation of MATLAB Data Types” on page 2-2.
  - 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.

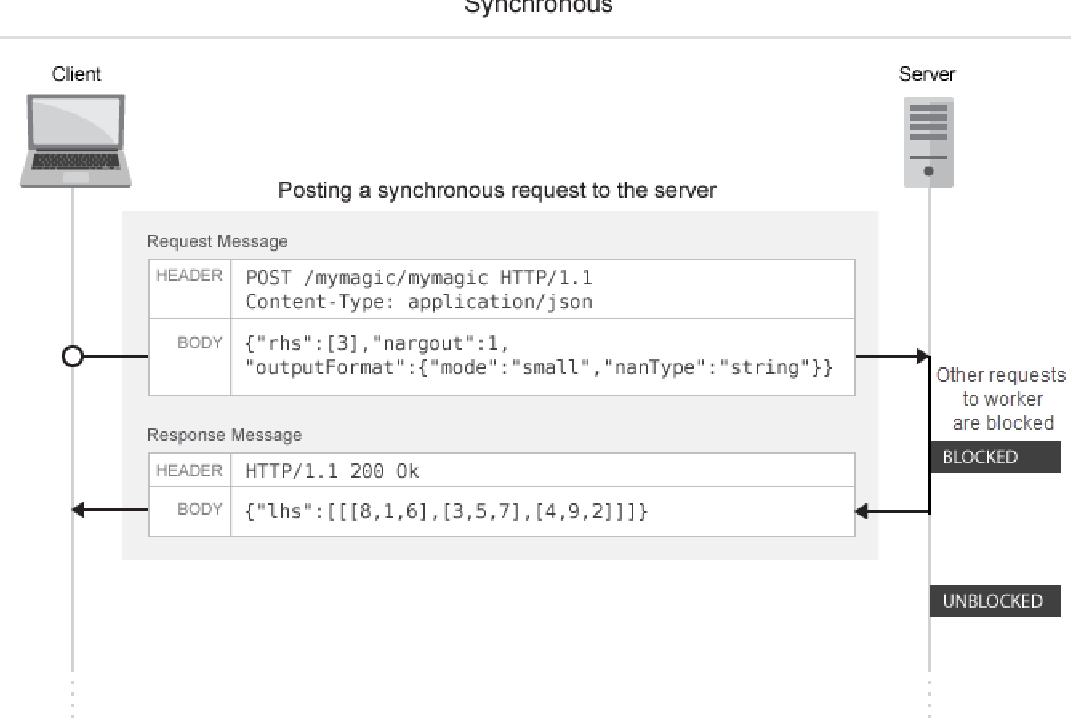
  If you use protobuf for data serialization, the Java and .NET client libraries provide methods and classes to deserialize the protobuf responses.
- The API supports both the synchronous and asynchronous modes of the server.

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

Synchronous Execution

In synchronous mode, after a client posts a request, the worker process of the server blocks all further requests until it has completed processing the original request. After processing is complete, the worker automatically returns a response to the client. Since it is the worker that blocks during request processing, if there are other workers available, the server can accept other synchronous requests for processing. To make a synchronous request to the server and wait for a response, use POST Synchronous Request.

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



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

This example shows how to use the RESTful API and JSON by providing two separate implementations—one using JavaScript on page 1-4 and the other using Python on page 1-4. 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);
end
```

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 Deployable Archive for MATLAB Production Server”. For more information on setting up a server, see “Create Server Instance Using Command Line”.

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`.

A sample HTML code with embedded JavaScript follows.

```
<!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

```
import json
import http.client

conn = http.client.HTTPConnection("localhost:9910")
headers = { "Content-Type": "application/json"}
body = json.dumps({"nargout": 1, "rhs" : [5]})
conn.request("POST", "/mymagic/mymagic", body, headers)
response = conn.getresponse()
if response.status == 200:
   result = json.loads(response.read())
   if "lhs" in result:
     print("Result of magic(5) is " + str(result["lhs"][0]["mwdata"]))
```

```
elif "error" in result:
  print("Error: " + str(result["error"]["message"]))
```

For an end-to-end workflow example of deploying a MATLAB function to MATLAB Production Server and invoking it using RESTful API and JSON, see “Create Web-Based Tool Using RESTful API, JSON, and JavaScript” on page 4-2.

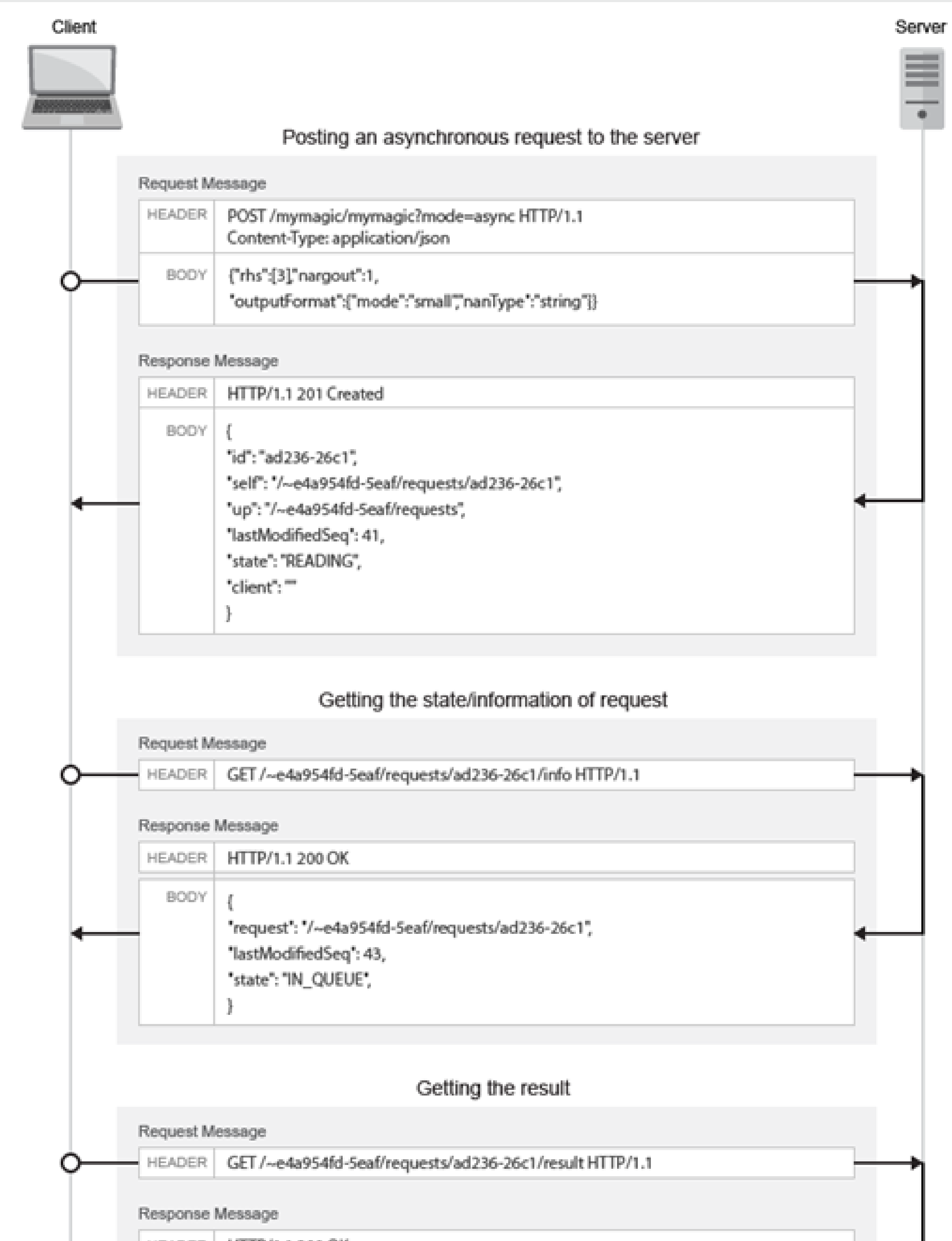
Asynchronous Execution

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 that the server returns for querying and retrieving results among other uses.

The RESTful API calls for asynchronous mode are listed in the following table:

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

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



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

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);
end
```

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 Deployable Archive for MATLAB Production Server”. For more information on setting up a server, see “Create Server Instance Using Command Line”.

A sample HTML code with embedded JavaScript follows.

```
<!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;
                            }
```

```
                                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>
```

Manage HTTP Cookie

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 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.

Asynchronous Request Execution

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.

Synchronous Request Execution

A client program that uses synchronous requests to execute a MATLAB function deployed to the server must not set the `Cookie` HTTP request header with the value of the `Set-Cookie` header, and must clear the value of the `Cookie` header if it has been previously set. This ensures that the synchronous requests are load balanced and the same server VM does not process them.

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”.

See Also

More About

- “RESTful API for Discovery and Diagnostics” on page 1-10
- “MATLAB Function Signatures in JSON” on page 1-18
- “JSON Representation of MATLAB Data Types” on page 2-2
- “Create Deployable Archive for MATLAB Production Server”

RESTful API for Discovery and Diagnostics

The MATLAB Production Server RESTful API for discovery and diagnostics consists of the following APIs:

- A discovery service that provides information about MATLAB functions deployed on a server
- A health check API that lets you know if the server is available to process requests
- A server metrics service that returns information about client requests, the time and memory that the server takes to execute these requests, and optional custom metrics

The health check and the discovery service return responses in JSON format. The metrics service returns data in Prometheus metrics format.

Characteristics of RESTful API

The MATLAB Production Server RESTful API for discovery and diagnostics uses the HTTP request-response model for communication with MATLAB Production Server. This model includes request methods, response codes, message headers, and message bodies. The RESTful API for discovery and diagnostics has the following characteristics:

- The HTTP GET method forms the primary mode of communication between client and server.
- Unique uniform resource identifiers (URIs) identify the resources that the server creates.
- Since requests to the server use the GET method, the requests do not have a message body and you do not have to set the `Content-Type` header in the request.
- The message body of the response contains information specific to a request such as information about functions deployed to the server, server health status, or server metrics.

Discovery Service

Use the discovery service to learn about 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.

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 Deployable Archive for MATLAB Production Server”. For information about creating the JSON file containing function signatures, see “MATLAB Function Signatures in JSON” on page 1-18.

Call the discovery service using GET Discovery Information.

The response from the server is a JSON object.

archives object

typedefs object

functions 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>" ], "mwsize": [ "<size1>, ..., <sizeN>" ], "help": "<field_description>"}
                ]
              }

              "<homogeneous_cell_name>":
              {
                "help": "<field_description>",
                "mwtype": "cell",
                "elements": {"name": "<field_name>", "mwtype": [ "<field_matlab_type>" ], "mwsize": [ "<size1>, ..., <sizeN>" ], "help": "<field_description>"}
              }

              "<heterogeneous_cell_name>":
              {
                "help": "<field_description>",
                "mwtype": "cell",
                "elements": [
                  {"name": "<field_name>", "mwtype": [ "<field_matlab_type>" ], "mwsize": [ "<size1>, ..., <sizeN>" ], "help": "<field_description>"}
                  {"name": "<field_name>", "mwtype": [ "<field_matlab_type>" ], "mwsize": [ "<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

The JSON response object contains a version number for the discovery schema and a list of deployed archives. The response object contains the following fields:

Key	Value
`discoverySchemaVersion`	JSON string containing the version number for the discovery schema in the format <major#>.<minor#>.<patch#>, where each number is a nonnegative integer *Example value:* `1.0.0`
`archives`	JSON object containing a list of all deployed archives

archives JSON Object

The `archives` object contains a list of all deployed archives. Each object in this list is a JSON object whose key is the name of the deployed archive, for example, `<Name of the CTF archive>`, and whose value is a JSON object that has the following fields:

Key	Value
`archiveSchemaVersion`	JSON string representing the version number of the archive schema *Example:* `1.1.0`
`archiveUuid`	JSON string representing a unique identifier for the archive
`matlabRuntimeVersion`	JSON string representing the MATLAB Runtime version *Example:* `9.9.0`
`functions`	JSON object containing a list of functions in the deployed archive
`typedefs`	JSON object containing a list of cell arrays or structures used as input or output arguments to deployed functions

functions JSON Object

The `functions` object contains a list of nested JSON objects, where each nested object corresponds to a MATLAB function in the deployed archive.

Each function object has the name of the deployed function as its key, for example, `<MATLAB_function_name1>`, and a JSON object as its value. The JSON object contains a `signatures` key whose value is an array of JSON objects that contain information about the MATLAB function signatures.

Each object in the `signatures` array contains the following fields:

Key	Value
`help`	Name of input parameter *Example:* `"name": "input1"`
`inputs`	Array of JSON objects containing information about input arguments
`outputs`	Array of JSON objects containing information about output arguments

Each object in the `inputs` array contains the following fields:

Key	Value
`name`	Name of input parameter *Example:* `"name": "input1"`

Key	Value
`mwtype`	MATLAB data type *Example:* `"mwtype": "double"`
`mwsize`	Size of data *Example:* `"mwsize": ["2,3"]`
`help`	Description for input arguments *Example:* `"help": "input1 description"`

Each object in the `outputs` object contains the following fields:

Key	Value
`name`	Name of output parameter *Example:* `"name": "output1"`
`mwtype`	MATLAB data type *Example:* `"mwtype": "double"`
`mwsize`	Size of data *Example:* `"mwsize": ["2,3"]`
`help`	Description for output parameters *Example:* `"help": "output1 description"`

typedefs JSON Object

The response contains the `typedefs` object only if deployed functions contain cell arrays or structures as input or output arguments.

If deployed functions contain cell arrays as input or output arguments, the `typedefs` object contains nested objects whose key is the name of the cell array, for example, `<homogeneous_cell_name>`, and the corresponding value contains an object with information about the cell array.

Each object in the `<cell_array_name>` object contains the following fields:

Key	Value
`help`	JSON string containing the description for the cell array *Example:* `"help": "cell help"`
`type`	`cell`
`elements`	JSON array of objects describing each element of the cell array

Each object in `elements` contains the following fields:

Key	Value
`name`	Name of cell element *Example:* `"name": "a"`
`type`	Data type of element *Example:* `"type": "double"`
`size`	Size of array *Example:* `"size": ["2,3"]`
`help`	Description of cell element *Example:* `"help": "Operand a"`

If deployed functions contain structure arrays as input or output arguments, the `typedefs` object contains nested objects whose key is the name of the structure, for example, `<struct_name>`, and the corresponding value contains an object with information about the structure.

Each object in the `<struct_name>` object contains the following fields:

Key	Value
`help`	JSON string containing the description for the structure *Example:* `"help": "struct help"`
`type`	`struct`
`fields`	JSON array of objects describing each element of the structure

Each object in `fields` contains the following fields:

Name	Description
`name`	Name of `struct` field *Example:* `"name": "my_field_name"`
`type`	Data type of field value *Example:* `"type": "char"`
`size`	Size of `struct` *Example:* `"size": ["2,3"]`
`help`	Description for `struct` element *Example:* `"help": "description for my_field_name"`

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. To check the server health, use GET Server Health.

A server is healthy when it is in one of the following states:

- 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.

If the health check is successful, the server responds with a `200 OK` HTTP status code and a JSON object indicating that the server is healthy.

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

When the server is unavailable to process HTTP requests, the health check API returns a `503 Health Check Failed` HTTP response code with an empty response body. The health check fails when the server has lost communication with the Network License Manager for a period of time exceeding the grace period. When the server is in this state, it actively attempts to reestablish communication with the license manager. Request processing resumes if the sever is able to reestablish communication with the license manager.

A failed health check 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. Your terminal must be on the same system as the server to run `mps-status`.

For more information on licensing, see “Manage Licenses for MATLAB Production Server”.

Metrics Service

Use the metrics service to retrieve server metrics in Prometheus® metrics format. The metrics service returns information about requests that client applications send to the server, and the time and memory that the server takes to execute the requests. You can use the metrics to monitor the server when working with Kubernetes® and microservices. To call the metrics services, use GET Metrics.

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

A successful response from the server consists of several server metrics in the Prometheus counter and gauge metric types. For more information about Prometheus metrics format, see Prometheus Metric Types.

```
# TYPE matlabprodserver_up_time_seconds counter
matlabprodserver_up_time_seconds 68140.5
# TYPE matlabprodserver_queue_time_seconds gauge
matlabprodserver_queue_time_seconds 0
# TYPE matlabprodserver_cpu_time_seconds counter
matlabprodserver_cpu_time_seconds 18.2188
# TYPE matlabprodserver_memory_working_set_bytes gauge
matlabprodserver_memory_working_set_bytes 1.57426e+08
# TYPE matlabprodserver_requests_accepted_total counter
```

```
matlabprodserver_requests_accepted_total 0
# TYPE matlabprodserver_requests_in_queue gauge
matlabprodserver_requests_in_queue 0
# TYPE matlabprodserver_requests_processing gauge
matlabprodserver_requests_processing 0
# TYPE matlabprodserver_requests_succeeded_total counter
matlabprodserver_requests_succeeded_total 0
# TYPE matlabprodserver_requests_failed_total counter
matlabprodserver_requests_failed_total 0
# TYPE matlabprodserver_requests_canceled_total counter
matlabprodserver_requests_canceled_total 0
```

An error response of `403 Metrics Disabled` indicates that the metrics service is not enabled on the server.

Custom Metrics

You can instrument deployed MATLAB code by adding custom metrics specific to your application or request processing. In the deployed MATLAB code, you can create custom Prometheus metrics by using the functions `prodserver.metrics.incrementCounter` and `prodserver.metrics.setGauge`. The functions create metrics of Prometheus counter and gauge metric types, respectively.

The server collects the custom metrics when a client calls the deployed MATLAB function. In addition to the default server metrics, the output of the metrics service includes the custom metrics and the name of the deployable archive that created the metrics.

For example, including the following functions in the MATLAB function that you deploy to the server creates custom metrics called `test_function_execution_count` and `test_timer_seconds`.

```
prodserver.metrics.incrementCounter("test_function_execution_count",1);
prodserver.metrics.setGauge("test_timer_seconds",0.421147);
```

When you query the metrics API after a client calls the deployed function, you see the following output:

```
# TYPE matlabprodserver_up_time_seconds counter
matlabprodserver_up_time_seconds 16705.3
# TYPE matlabprodserver_queue_time_seconds gauge
matlabprodserver_queue_time_seconds 0
# TYPE matlabprodserver_cpu_time_seconds counter
matlabprodserver_cpu_time_seconds 29.1406
# TYPE matlabprodserver_memory_working_set_bytes gauge
matlabprodserver_memory_working_set_bytes 5.17153e+08
# TYPE matlabprodserver_requests_accepted_total counter
matlabprodserver_requests_accepted_total 7
# TYPE matlabprodserver_requests_in_queue gauge
matlabprodserver_requests_in_queue 0
# TYPE matlabprodserver_requests_processing gauge
matlabprodserver_requests_processing 0
# TYPE matlabprodserver_requests_succeeded_total counter
matlabprodserver_requests_succeeded_total 7
# TYPE matlabprodserver_requests_failed_total counter
matlabprodserver_requests_failed_total 0
# TYPE matlabprodserver_requests_canceled_total counter
matlabprodserver_requests_canceled_total 0
# TYPE test_function_execution_count counter
```

```
test_function_execution_count{archive="test_metrics_2"} 1
# TYPE test_timer_seconds gauge
test_timer_seconds{archive="test_metrics"} 0.421147
```

The output contains the `test_function_execution_count` and `test_timer_seconds` custom metrics, and the name of the deployable archive, `test_metrics`, that generates the metrics.

For a detailed example, see "Create Custom Prometheus Metrics" on page 4-9.

See Also

`mps-status` | `license-grace-period` | `prodserver.metrics.setGauge` | `prodserver.metrics.incrementCounter`

Related Examples

- "RESTful API for MATLAB Function Execution" on page 1-2
- "Verify Server Status"
- "Create Custom Prometheus Metrics" on page 4-9

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#>",
                                                                    Typedef Object
  "_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"}
    }
  },
                                                Function Object
  "functionName1":
  {                                                        Signature Object
    "inputs":
    [                                                                          Argument Object
      {"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#>`, where 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
`"int8"`	Must be an 8-bit signed integer

Value	Argument Description
"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 1-21.

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 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 `[]` .
`"mwsize"`	JSON `array` representing the dimensions of the data. Specify the property value by enclosing the dimensions as a comma-separated list within `[]`.

`"mwtype"`	JSON `string` representing the type of data. Specify the property value within `""`.
	`"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 `true`.

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

Function Name	Purpose
`mps.json.encoderequest`	Convert MATLAB data in a server request to JSON text using MATLAB Production Server JSON schema.
`mps.json.decoderesponse`	Convert JSON text from a server response to MATLAB data.
`mps.json.encode`	Convert MATLAB data to JSON text using MATLAB Production Server JSON schema.
`mps.json.decode`	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 `mwdata` property must be a JSON array of JSON numbers.
- The `mwtype` 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 `mwdata` property must be a JSON array containing one JSON number representing the MATLAB scalar value.
- The `mwsize` 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

`single`, `int8`, `uint8`, `int16`, `uint16`,
 	
`int32`, `uint32`, `int64`, `uint64`
	No small representation	`{`
		`"mwdata": [JSON number],`
		`"mwsize": [1,1],`
		`"mwtype": "single" | "int8" | "uint`
		`"int32" | "uint32" | "int6`
		`}`

`double`	JSON number	`{`
		`"mwdata": [JSON number],`
		`"mwsize": [1,1],`
		`"mwtype": "double"`
		`}`

MATLAB Data: Scalar Numerics	JSON Small Notation	JSON Large Notation
`int8(23)`	No small representation	`{ "mwdata": [23], "mwsize": [1,1], "mwtype": "int8" }`
`uint8(27)`	No small representation	`{ "mwdata": [27], "mwsize": [1,1], "mwtype": "uint8" }`
`single(20.15)`	No small representation	`{ "mwdata": [20.15], "mwsize": [1,1], "mwtype": "single" }`
`intmax('int64')`	No small representation	`{ "mwdata": [9223372036854775807], "mwsize": [1,1], "mwtype": "int64" }`
`double(12.905)`	`12.905`	`{ "mwdata": [12.905], "mwsize": [1,1], "mwtype": "double" }`
`42`	`42`	`{ "mwdata": [42], "mwsize": [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], "mwsize":[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",`
		`    "mwsize":[2,3],`
		`    "mwtype":"double"`
		`}`

`[ 1 2; 4 5; 7 8 ]`	`[[1, 2], [4, 5], [7, 8]]`	`{`
		`    "mwdata":[1,4,7,2,5,8],`
		`    "mwsize":[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":[1,3,5,2,4,6,7,9,11,8,10,1`
		`    "mwsize":[3,2,2],`
		`    "mwtype":"double"`
		`}`

`[17;500]`	`[17,500]`	`{`
		`    "mwdata":[17,500],`
		`    "mwsize":[2,1],`
		`    "mwtype":"double"`
		`}`

`[17,500]`	`[[17,500]]`	`{`
		`    "mwdata":[17,500],`
		`    "mwsize":[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.

<table>
<tr><th>MATLAB Data: NaN, Inf, and -Inf</th><th>JSON Small Notation</th><th>JSON Large Notation</th></tr>
<tr><td>NaN</td><td>{"mwdata":"NaN"}</td><td><pre>{
    "mwdata": ["NaN"],
    "mwsize": [1,1],
    "mwtype": "double"
}</pre>or<pre>{
    "mwdata": [{"mwdata":"NaN"}],
    "mwsize": [1,1],
    "mwtype": "double"
}</pre></td></tr>
<tr><td>Inf</td><td>{"mwdata":"Inf"}</td><td><pre>{
    "mwdata": ["Inf"],
    "mwsize": [1,1],
    "mwtype": "double"
}</pre>or<pre>{
    "mwdata": [{"mwdata":"Inf"}],
    "mwsize": [1,1],
    "mwtype": "double"
}</pre></td></tr>
<tr><td>-Inf</td><td>{"mwdata":"-Inf"}</td><td><pre>{
    "mwdata": ["-Inf"],
    "mwsize": [1,1],
    "mwtype": "double"
}</pre>or<pre>{
    "mwdata": [{"mwdata":"-Inf"}],
    "mwsize": [1,1],
    "mwtype": "double"
}</pre></td></tr>
</table>

Numeric Types: Complex Numbers

- The `mwdata` 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

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

MATLAB Data: Scalar Complex Number	JSON Large Notation

`int32(3 + 4i)`	`{`
	`"mwcomplex": true,`
	`"mwdata": [3,4],`
	`"mwsize": [1,1],`
	`"mwtype": "int32"`
	`}`

MATLAB Data: Multidimensional Array of Complex Numbers	JSON Large Notation

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

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

`char`	`JSON string`	`{`
		`"mwdata": [JSON string],`
		`"mwsize": [<char dimensions>],`
		`"mwtype": "char"`
		`}`

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": [2,6],`
	`"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

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

MATLAB Data: Scalar `logical`	JSON Small Notation	JSON Large Notation

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

`logical(0)` or `false`	`false`	`{`
		`"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,fal`
		`    "mwsize": [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>],`
	`    "mwsize": [<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

`{'Primes',[10 23 199],{false,true,'maybe'}}`	`{`
	`"mwdata": ["Primes",`
	`[[10, 23, 199]],`
	`{`
	`"mwdata": [false, true, "maybe"],`
	`"mwsize": [1, 3],`
	`"mwtype": "cell"`
	`}`
	`],`
	`"mwsize": [1, 3],`
	`"mwtype": "cell"`
	`}`
		`{`
		`"mwdata": [{`
		`"mwdata": ["Primes"],`
		`"mwsize": [1, 6],`
		`"mwtype": "char"`
		`},{`
		`"mwdata": [10, 23, 199],`
		`"mwsize": [1, 3],`
		`"mwtype": "double"`
		`},{`
		`"mwdata": [{`
		`"mwdata": [false],`
		`"mwsize": [1, 1],`
		`"mwtype": "logical"`
		`},{`
		`"mwdata": [true],`
		`"mwsize": [1, 1],`
		`"mwtype": "logical"`
		`},{`
		`"mwdata": ["maybe"],`
		`"mwsize": [1, 5],`
		`"mwtype": "char"`
		`}],`
		`"mwsize": [1, 3],`
		`"mwtype": "cell"`
		`}],`
		`"mwsize": [1, 3],`
		`"mwtype": "cell"`
		`}`

- 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

`struct`	JSON object	`{`
		`"mwdata": {<struct data>}`
		`"mwsize": [<struct dimensions>],`
		`"mwtype": "struct"`
		`}`

-

MATLAB Data: Scalar Structure Array	JSON Small Notation	JSON Large Notation

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

- 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

`struct('Name',{{'Casper','Ghost'}},...`
`'Age',{[14,17,18]},...`
`'Date', {736676})`
	`{`
	`"Age": [[14, 17, 18]],`
	`"Date": 736676,`
	`"Name": {`
	`"mwdata": ["Casper", "Ghost"],`
	`"mwsize": [1, 2],`
	`"mwtype": "cell"`
	`}`
	`}`
		`{`
		`"mwdata": {`
		`"Age": [{`
		`"mwdata": [14, 17, 18],`
		`"mwsize": [1, 3],`
		`"mwtype": "double"`
		`}],`
		`"Date": [{`
		`"mwdata": [736676],`
		`"mwsize": [1, 1],`
		`"mwtype": "double"`
		`}],`
		`"Name": [{`
		`"mwdata": [{`
		`"mwdata": ["Casper"],`
		`"mwsize": [1, 6],`
		`"mwtype": "char"`
		`}, {`
		`"mwdata": ["Ghost"],`
		`"mwsize": [1, 5],`
		`"mwtype": "char"`
		`}],`
		`"mwsize": [1, 2],`
		`"mwtype": "cell"`
		`}]`
		`},`
		`"mwsize": [1, 1],`
		`"mwtype": "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.

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

`struct( 'Name',{'Casper','Ghost';...`
`'Genie' ,'Wolf'},...`
`'Ages',{14,17;...`
`20,23} )`
	`{`
	`"mwdata":{`
	`"Ages":[14,20,17,23],`
	`"Name":["Casper","Genie","Ghost","Wolf"]`
	`},`
	`"mwsize":[2,2],`
	`"mwtype":"struct"`
	`}`
		`{`
		`"mwdata":{`
		`"Ages":[{`
		`"mwdata":[14],`
		`"mwsize":[1,1],`
		`"mwtype":"double"`
		`},{`
		`"mwdata":[20],`
		`"mwsize":[1,1],`
		`"mwtype":"double"`
		`},{`
		`"mwdata":[17],`
		`"mwsize":[1,1],`
		`"mwtype":"double"`
		`},{`
		`"mwdata":[23],`
		`"mwsize":[1,1],`
		`"mwtype":"double"`
		`}],`
		`"Name":[{`
		`"mwdata":["Casper"],`
		`"mwsize":[1,6],`
		`"mwtype":"char"`
		`},{`
		`"mwdata":["Genie"],`
		`"mwsize":[1,5],`
		`"mwtype":"char"`
		`},{`
		`"mwdata":["Ghost"],`
		`"mwsize":[1,5],`
		`"mwtype":"char"`
		`},{`
		`"mwdata":["Wolf"],`
		`"mwsize":[1,4],`
		`"mwtype":"char"`
		`}]`
		`},`
		`"mwsize":[2,2],`
		`"mwtype":"struct"`
		`}`

- 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],`
	`"mwsize": [<string dimensions>],`
	`"mwtype": "string"`
	`}`

MATLAB Data: Scalar, Single-dimensional, Multidimensional, and `missing` (MATLAB) `string` Arrays	JSON Large Notation

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

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

`["abc" "de"]`	`{`
	`"mwdata": ["abc", "de"],`
	`"mwsize": [1, 2],`
	`"mwtype": "string"`
	`}`

`["abc" "de"; "fg" "hi"]`	`{`
	`"mwdata": ["abc", "fg", "de", "hi"],`
	`"mwsize": [2, 2],`
	`"mwtype": "string"`
	`}`

`string(missing)`	`{`
	`"mwdata": [{"mwdata": "missing"}],`
	`"mwsize": [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],
	"mwsize": [<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"],
	"mwsize": [1, 1],
	"mwtype": "Colors"
	}

b = [Colors.Black Colors.Blue]	{
	"mwdata": ["Black", "Blue"],
	"mwsize": [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 `NaT` 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

`datetime`	{
	   "mwdata": {
	                "LowOrderTimeStamp": <JSON number>
	                "TimeStamp": <JSON number>
	            },
	    "mwmetadata": {
	        "TimeZone": <JSON string>,
	        "Format": <JSON string>
	    },
	    "mwsize": [<datetime array dimensions>],
	    "mwtype": "datetime"
	}

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

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 `mwmetadata` in Small Notation**	**JSON Large Notation with `mwdata` and `mwmetadata` in Large Notation**

`datetime(2018,1,8,10,...`
`11,12,(1:5)+(1:5)*1e-6,...`
`'TimeZone','local');`
	`{`
	`"mwdata": {`
	`"LowOrderTimeStamp": [[9.9999999991773336E-7,`
	`1.9999999998354667E-6,`
	`2.9999999999752447E-6,`
	`3.9999999996709334E-6,`
	`4.9999999998107114E-6]],`
	`"TimeStamp": [[1.515424272001E+12,`
	`1.515424272002E+12,`
	`1.515424272003E+12,`
	`1.515424272004E+12,`
	`1.515424272005E+12]],`
	`},`
	`"mwmetadata": {`
	`"Format":"dd-MMM-uuuu HH:mm:ss",`
	`"TimeZone": "America\/New_York",`
	`},`
	`"mwsize": [1, 5],`
	`"mwtype": "datetime"`
	`}`
		`{`
		`"mwdata": {`
		`"LowOrderTimeStamp": {`
		`"mwdata": [9.999999999177`
		`1.9999999998354667E-6`
		`2.9999999999752447E-6`
		`3.9999999996709334E-6`
		`4.9999999998107114E-6`
		`"mwsize": [1, 5],`
		`"mwtype": "double"`
		`},`
		`"TimeStamp": {`
		`"mwdata": [1.515424272001`
		`1.515424272002E+12,`
		`1.515424272003E+12,`
		`1.515424272004E+12,`
		`1.515424272005E+12],`
		`"mwsize": [1, 5],`
		`"mwtype": "double"`
		`}`
		`},`
		`"mwmetadata": {`
		`"Format": {`
		`"mwdata": ["dd-MMM-uu`
		`"mwsize": [1, 20],`
		`"mwtype": "char"`
		`},`
		`"TimeZone": {`
		`"mwdata": ["America\/`
		`"mwsize": [1, 16],`
		`"mwtype": "char"`
		`}`
		`},`
		`"mwsize": [1, 5],`
		`"mwtype": "datetime"`
		`}`

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

`datetime(2018,1,8,10,...`
`11,12,(1:5)+(1:5)*1e-6,...`
`'TimeZone','local')';`
	`{`
	`    "mwdata": {`
	`        "LowOrderTimeStamp": [9.999999999177336E-7,`
	`            1.9999999998354667E-6,`
	`            2.9999999999752447E-6,`
	`            3.9999999996709334E-6,`
	`            4.9999999998107114E-6],`
	`        "TimeStamp": [1.515424272001E+12,`
	`            1.515424272002E+12,`
	`            1.515424272003E+12,`
	`            1.515424272004E+12,`
	`            1.515424272005E+12],`
	`    },`
	`    "mwmetadata": {`
	`        "Format":"dd-MMM-uuuu HH:mm:ss",`
	`        "TimeZone": "America\/New_York",`
	`    },`
	`    "mwsize": [5, 1],`
	`    "mwtype": "datetime"`
	`}`
		`{`
		`    "mwdata": {`
		`        "LowOrderTimeStamp": {`
		`            "mwdata": [9.999999999177`
		`                1.9999999998354667E-6`
		`                2.9999999999752447E-6`
		`                3.9999999996709334E-6`
		`                4.9999999998107114E-6`
		`            "mwsize": [1, 5],`
		`            "mwtype": "double"`
		`        },`
		`        "TimeStamp": {`
		`            "mwdata": [1.515424272001`
		`                1.515424272002E+12,`
		`                1.515424272003E+12,`
		`                1.515424272004E+12,`
		`                1.515424272005E+12],`
		`            "mwsize": [5, 1],`
		`            "mwtype": "double"`
		`            }`
		`        },`
		`        "mwmetadata": {`
		`            "Format": {`
		`                "mwdata": ["dd-MMM-uu`
		`                "mwsize": [1, 20],`
		`                "mwtype": "char"`
		`            },`
		`            "TimeZone": {`
		`                "mwdata": ["America\/`
		`                "mwsize": [1, 16],`
		`                "mwtype": "char"`
		`        }`
		`    },`
		`    "mwsize": [1, 5],`
		`    "mwtype": "datetime"`
		`}`

MATLAB Data: NaT and Inf datetime Array	JSON Large Notation with mwdata and mwmetadata in Small Notation	JSON Large Notation with mwdata and mwmetadata in Large Notation

NaT	{
	"mwdata": {
	"TimeStamp": {
	"mwdata": "NaN"
	}
	},
	"mwmetadata": {
	"Format": "dd-MMM-uuuu HH:mm:ss",
	"TimeZone": ""
	},
	"mwsize": [1, 1],
	"mwtype": "datetime"
	}
		{
		"mwdata": {
		"TimeStamp": {
		"mwdata": ["NaN"],
		"mwsize": [1, 1],
		"mwtype": "double"
		}
		},
		"mwmetadata": {
		"Format": {
		"mwdata": ["dd-MMM-uuuu H
		"mwsize": [1, 20],
		"mwtype": "char"
		},
		"TimeZone": {
		"mwdata": [""],
		"mwsize": [0, 0],
		"mwtype": "char"
		}
		},
		"mwsize": [1, 1],
		"mwtype": "datetime"
		}

datetime(inf,inf,inf)	{
	"mwdata": {
	"TimeStamp": {
	"mwdata": "Inf"
	}
	},
	"mwmetadata": {
	"Format": "dd-MMM-uuuu HH:mm:ss",
	"TimeZone": ""
	},
	"mwsize": [1, 1],
	"mwtype": "datetime"
	}
		{
		"mwdata": {
		"TimeStamp": {
		"mwdata": ["Inf"],
		"mwsize": [1, 1],
		"mwtype": "double"
		}
		},
		"mwmetadata": {
		"Format": {
		"mwdata": ["dd-MMM-uu
		"mwsize": [1, 20],
		"mwtype": "char"
		},
		"TimeZone": {
		"mwdata": [""],
		"mwsize": [0, 0],
		"mwtype": "char"
		}
		},
		"mwsize": [1, 1],
		"mwtype": "datetime"
		}

- 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

[]	[]	{
		"mwdata": [],
		"mwsize": [0,0],
		"mwtype": "double" | "single"
		"int8" | "uint8" | "i
		"int32" | "uint32" | "i
		"logical" | "char" | "c
		"<class name of enumeratio
		}

See Also

More About

- "RESTful API for MATLAB Function Execution" 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. For more information, see "Structure of MATLAB Error" on page 3-4. To review RESTful API usage, see "RESTful API for MATLAB Function Execution" on page 1-2.

Structure of HTTP Error

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

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 do not follow the JSON representation for MATLAB data types. For more information, see "JSON Representation of MATLAB Data Types" on page 2-2.

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 is unable to find the MATLAB function in the deployed CTF archive
`Component not found`	Server is unable to find the CTF archive
`URI-path not of form '/APPLICATION/FUNCTION'`	URL is 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`	Content type for JSON is incorrect.

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

To resolve a MATLAB error, troubleshoot the MATLAB function deployed on the server.

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

Access-Control-Allow-Origin

Client programmers using JavaScript need to verify whether Cross-Origin Resource Sharing (CORS) is enabled on a MATLAB Production Server instance, if their clients programs make 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 for MATLAB Function Execution" on page 1-2

Examples: RESTful API and JSON

Create 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 `pricecalc.m`.

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

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

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

To create the deployable archive for this example:

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

5 Click **Package**.

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

Step 3: Place the Deployable Archive on a Server

1. Download the MATLAB Runtime, if needed, at https://www.mathworks.com/products/compiler/mcr. See “Supported MATLAB Runtime Versions for MATLAB Production Server” for more information.
2. Create a server using `mps-new`. See “Create Server Instance Using Command Line” 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 “Configure Server Using 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 “Configure Server Using 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.statusTex
            document.getElementById("response").innerHTML = "Status: " + request.status + "<
                    + "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>
      <hr>
      <h3>M: Face Value </h3>
      <input id="facevalue_value" type="number" maxlength="4" oninput="updatePrice('facevalue_va
      <input type="range" id="facevalue_slider" value="0" min="0" max="10000" onchange="updatePri

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

      <h3>N: Number of payments  </h3>
      <input id="num_payments_value" type="number" maxlength="4" oninput="updatePrice('num_paymen
      <input type="range" id="num_payments_slider" value="0" min="0" max="1000" onchange="update

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

    <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

1000

C: Coupon Payment

100

N: Number of payments

5

i: Interest rate

0.08

BOND PRICE

$: 1079.8542007415617

Request to MPS Server

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

Response from MPS Server

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

See Also

More About

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

Create Custom Prometheus Metrics

This example shows how to create custom Prometheus metrics on a MATLAB Production Server instance and retrieve them using the metrics service. To create custom metrics, use the `prodserver.metrics.setGauge` and `prodserver.metrics.incrementCounter` functions in the deployed MATLAB function. After you execute the deployed function, query the "Metrics Service" on page 1-15 to retrieve the custom metrics.

The example assumes that there is an on-premises server running at `http://localhost:9910` managed using the command line.

Write MATLAB Code to Create Custom Metrics

Write a MATLAB function that calls the `proprodserver.metrics.setGauge` and `prodserver.metrics.incrementCounter` functions to create Prometheus metrics of type gauge and counter, respectively. `proprodserver.metrics.setGauge` and `prodserver.metrics.incrementCounter` do not return any output.

The following `test_metrics` function only creates example metrics. In practice, the MATLAB programmer, in consultation with the server administrator, creates metrics related to the deployed application that help instrument the deployed MATLAB code.

```
function rc = test_metrics()
tic
prodserver.metrics.incrementCounter("test_function_execution_count",1)
toc
prodserver.metrics.setGauge("test_timer_seconds",toc)
rc = 0;
end
```

Deploy MATLAB Function to Server

Package the function `test_metrics.m` into a deployable archive called `test_metrics` and deploy it to the server.

For details on creating and starting a server, see "Create Server Instance Using Command Line" and "Start Server Instance Using Command Line".

For details on creating a deployable archive and deploying it to the server, see "Create Deployable Archive for MATLAB Production Server" and "Deploy Archive to MATLAB Production Server".

Enable Metrics on Server

Enable the metrics service on the server by editing the server configuration file, `main_config`. In `main_config`, uncomment the `--enable-metrics` property. Restart the server for the changes to take effect.

Execute Deployed Function

Using the language of your choice, write a client application to execute the deployed function. This example uses a cURL command on a Windows® terminal to send requests to the server. For more

information on constructing requests, see “JSON Representation of MATLAB Data Types” on page 2-2.

The following command executes the function `test_metrics` deployed to a server running at `http://localhost:9910`. Executing the function increments the `test_function_execution_count` metric by 1 and sets the `test_timer_seconds` metric to a varying number.

```
curl -v -H Content-Type:application/json -d '{"nargout":0,"rhs":[]}' http://localhost:9910/test_metrics/test_metrics
```

Query Metrics Service to Retrieve Custom Metrics

Custom metrics are registered on the server after a client calls the deployed MATLAB function. To retrieve the metrics, use the GET Metrics API by accessing the following URL in a web browser. In practice, a Prometheus server scrapes the metrics HTTP/HTTPS endpoint.

```
http://localhost:9910/api/metrics
```

The output of the metrics service contains information about server metrics. It also contains the `test_function_execution_count` and `test_timer_seconds` custom metrics, along with the name of the deployable archive, `test_metrics`, that generates the metrics.

```
# TYPE matlabprodserver_up_time_seconds counter
matlabprodserver_up_time_seconds 16705.3
# TYPE matlabprodserver_queue_time_seconds gauge
matlabprodserver_queue_time_seconds 0
# TYPE matlabprodserver_cpu_time_seconds counter
matlabprodserver_cpu_time_seconds 29.1406
# TYPE matlabprodserver_memory_working_set_bytes gauge
matlabprodserver_memory_working_set_bytes 5.17153e+08
# TYPE matlabprodserver_requests_accepted_total counter
matlabprodserver_requests_accepted_total 7
# TYPE matlabprodserver_requests_in_queue gauge
matlabprodserver_requests_in_queue 0
# TYPE matlabprodserver_requests_processing gauge
matlabprodserver_requests_processing 0
# TYPE matlabprodserver_requests_succeeded_total counter
matlabprodserver_requests_succeeded_total 7
# TYPE matlabprodserver_requests_failed_total counter
matlabprodserver_requests_failed_total 0
# TYPE matlabprodserver_requests_canceled_total counter
matlabprodserver_requests_canceled_total 0
# TYPE test_function_execution_count counter
test_function_execution_count{archive="test_metrics_2"} 1
# TYPE test_timer_seconds gauge
test_timer_seconds{archive="test_metrics_2"} 0.0194095
```

Since the metric type of `test_function_execution_count` is a counter, its value increases by 1 every time you execute the deployed function and query the metrics service. Since the metric type of `test_timer_seconds` is a gauge, its value can increase or decrease every time you execute the deployed function and query the metrics service.

See Also

`prodserver.metrics.setGauge` | `prodserver.metrics.incrementCounter`

Related Examples

- “RESTful API for MATLAB Function Execution” on page 1-2
- “Metrics Service” on page 1-15

- GET Metrics

External Websites

- Prometheus Metric Types

5

RESTful APIs

POST Synchronous Request

Make synchronous request to server, and wait for response

Description

Use a POST method to make a synchronous request to the server. In synchronous mode, after the server receives the request, the worker process on the server blocks all further requests until it has completed processing the original request. The worker automatically returns a response to the client after processing is complete. No other HTTP methods are necessary to retrieve the response from the server.

The server can simultaneously execute as many synchronous requests as the number of available workers.

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
`nargout`	Number of outputs that the client application is requesting from the deployed MATLAB function. Note that MATLAB functions, depending on their intended purpose, can be coded to return multiple outputs. A subset of these potential outputs can be specified using `nargout`.	`number`

Name	Description	Value-Type
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:

```
{
 "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);
```

Version History

Introduced in R2016a

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”

POST Asynchronous Request

Make asynchronous request to 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. Use the URI in other asynchronous execution requests such as retrieving the state of the request or result of request.	`{request-uri-string}`
`up`	URI of a collection of requests tied to a particular client.	`{request-collection-uri-string}`
`lastModifiedSeq`	Number indicating when a request represented by `self` was last modified.	`{server-state-number}`

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

`client`	Client id or 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);
```

Version History

Introduced in R2016b

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 1-7
“Create 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

GET Representation of Asynchronous Request

View how asynchronous request made to 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
`id`	ID of a particular request.	`{id-string}`
`self`	URI of particular request. Use the URI in other asynchronous execution requests such as retrieving the state of the request or result of request.	`{request-uri-string}`
`up`	URI of a collection of requests tied to a particular client.	`{request-collection-uri-string}`
`lastModifiedSeq`	Number indicating when a request represented by `self` was last modified.	`{server-state-number}`

Name	Description	Value-Type
state	State of a request.	{request-state-string} Possible states: READING IN_QUEUE PROCESSING READY ERROR CANCELLED
client	Client id or name that was specified as a query parameter while initiating an asynchronous request.	{client-id-string}

Example:


```
{
  "id": "f90c2ff8-4d27-4795-806d-18c351abeb5b",
  "self": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/f90c2ff8-4d27-4795-806d-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.
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);
```

Version History

Introduced in R2016b

See Also

GET State Information | GET Result of Request

Topics

“Asynchronous Execution” on page 1-5
“Asynchronous RESTful Requests Using Protocol Buffers in the Java Client”

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.	`"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}`
		`},...`
		`]`

Example:

```
"data": [
    {
      "id": "c5666088-b087-4bae-aa7d-d8470e6e082d",
      "self": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/c5666088-b087-4bae-aa7d-d8470e6e0
      "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-a855-6d6419284
      "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-d8470e6e0
      "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-a855-6d6419284
      "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);
```

Version History

Introduced in R2016b

See Also

GET State Information | GET Representation of Asynchronous Request

Topics

"Asynchronous RESTful Requests Using Protocol Buffers in the Java Client"

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

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

Example:

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

Response:

```
Status Code: 200 OK
{
  "request": "/~e4a954fd-5eaf-4b54-aac2-20681b33d075/requests/c5666088-b087-4bae-aa7d-d8470e6e08
  "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-b
xhr.send(data);
```

Version History

Introduced in R2016b

See Also

GET Representation of Asynchronous Request | GET Result of Request

Topics

“Asynchronous Execution” on page 1-5

"Asynchronous RESTful Requests Using Protocol Buffers in the Java Client"

GET Result of Request

Retrieve results of 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);
```

Version History

Introduced in R2016b

See Also

GET State Information | DELETE Request

Topics

"Asynchronous Execution" on page 1-5
"Asynchronous RESTful Requests Using Protocol Buffers in the Java Client"

POST Cancel Request

Cancel request

Description

Use a POST method to cancel a request. You can cancel only those requests that have not already completed.

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);
```

Version History

Introduced in R2016b

See Also

DELETE Request | POST Asynchronous Request

Topics

"Asynchronous Execution" on page 1-5

DELETE Request

Delete request from server

Description

Use a DELETE method to delete a request on the server. You cannot retrieve the information of a deleted request.

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
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);
```

Version History

Introduced in R2016b

See Also

POST Cancel Request | POST Asynchronous Request

Topics

"Asynchronous Execution" on page 1-5

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",
                                    "mwsize": [],
                                    "help": "Dimension of magic square matrix"
                                }
                            ],
                            "outputs": [
                                {
                                    "name": "out",
                                    "mwtype": "double",
                                    "mwsize": [],
                                    "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);
```

Version History

Introduced in R2018a

See Also

Topics

“Discovery Service” on page 1-10
“MATLAB Function Signatures in JSON” on page 1-18

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);
```

Version History

Introduced in R2019b

See Also

Topics

“Health Check” on page 1-14

GET Metrics

Retrieve server metrics

Description

Use the GET method to retrieve metrics for a server instance in the Prometheus metrics format. The metrics service returns information about requests that client applications send to the server, and the time and memory that the server takes to execute the requests. You can use the metrics service to monitor server metrics in a Kubernetes environment. All server metrics reset on a server restart.

In addition to the server metrics, the metrics service also returns custom metrics that you can create in the MATLAB function that you deploy to the server. For a detailed example, see “Create Custom Prometheus Metrics” on page 4-9. Custom metrics reset depending on the value of the worker-restart-interval property.

To use the metrics service, you must enable the metrics service on the server. Do this by uncommenting the option `--enable-metrics` in the `main_config` server configuration file.

Request

HTTP Method

`GET`

URI

`http(s)://host:port/api/metrics`

Response

Success

HTTP Status Code

`200 OK`

Body

Name	Description
`matlabprodserver_up_time_seconds`	Time in fractional seconds since server startup.
`matlabprodserver_queue_time_seconds`	Sum of wait times in fractional seconds for currently queued synchronous and asynchronous requests.
`matlabprodserver_cpu_time_seconds`	Total CPU time in fractional seconds that the server spent in request execution after startup.
`matlabprodserver_memory_working_set_bytes`	Sum of memory utilization in bytes by all MATLAB Production Server processes at a given time.

Name	Description
matlabprodserver_requests_accepted_total	Total number of valid requests that the server received after startup. Total requests accepted at a given time is the sum of requests that are canceled, in queue, processing, and requests that have failed and successfully completed after server startup.
matlabprodserver_requests_in_queue	Number of requests currently waiting to be processed by the server.
matlabprodserver_requests_processing	Number of requests that the server is currently processing.
matlabprodserver_requests_succeeded_total	Total number of requests that completed successfully.
matlabprodserver_requests_failed_total	Total number of requests that failed. Requests can fail if they contain an incorrect name of the deployed MATLAB function.
matlabprodserver_requests_canceled_total	Total number of asynchronous requests that clients canceled.

Error

`403 Metrics Disabled`

Sample Call

HTTP

Request:

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

Response:

```
# TYPE matlabprodserver_up_time_seconds counter
matlabprodserver_up_time_seconds 68140.5
# TYPE matlabprodserver_queue_time_seconds gauge
matlabprodserver_queue_time_seconds 0
# TYPE matlabprodserver_cpu_time_seconds counter
matlabprodserver_cpu_time_seconds 18.2188
# TYPE matlabprodserver_memory_working_set_bytes gauge
matlabprodserver_memory_working_set_bytes 1.57426e+08
# TYPE matlabprodserver_requests_accepted_total counter
matlabprodserver_requests_accepted_total 0
# TYPE matlabprodserver_requests_in_queue gauge
matlabprodserver_requests_in_queue 0
# TYPE matlabprodserver_requests_processing gauge
matlabprodserver_requests_processing 0
# TYPE matlabprodserver_requests_succeeded_total counter
matlabprodserver_requests_succeeded_total 0
# TYPE matlabprodserver_requests_failed_total counter
matlabprodserver_requests_failed_total 0
# TYPE matlabprodserver_requests_canceled_total counter
matlabprodserver_requests_canceled_total 0
```

JavaScript

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

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

Version History

Introduced in R2021a

See Also

`mps-status` | `prodserver.metrics.setGauge` | `prodserver.metrics.incrementCounter`

Topics

"Metrics Service" on page 1-15

External Websites

Prometheus Metric Types