Creating REST Services

Version 2023.1
2024-07-11

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Creating REST Services
InterSystems IRIS Data Platform Version 2023.1 2024-07-11

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

1

Introduction to Creating REST Services

This page introduces REST and REST services in InterSystems IRIS® data platform. You can use these REST interfaces with UI tools, such as Angular, to provide access to databases and interoperability productions. You can also use them to enable external systems to access InterSystems IRIS applications. For an interactive introduction to REST services, try Developing REST Interfaces.

1.1 Introduction to REST

REST, which is named from "Representational State Transfer," has the following attributes:

- REST is an architectural style rather than a format. Although REST is frequently implemented using HTTP for transporting messages and JSON for passing data, you can also pass data as XML or plain text. REST makes use of existing web standards such as HTTP, URL, XML, and JSON.
- REST is resource oriented. Typically a resource is identified by a URL and uses operations based explicitly on HTTP methods, such as GET, POST, PUT, DELETE, and OPTIONS.
- REST typically has a small overhead. Although it can use XML to describe data, it more commonly uses JSON which is a lightweight data wrapper. JSON identifies data with tags but the tags are not specified in a formal schema definition and do not have explicit data types.

1.2 Introduction to InterSystems REST Services

There are two ways to define REST interfaces in InterSystems IRIS 2019.2 and later:

- Specification-first definition — you first create an OpenAPI 2.0 specification and then use the API Management tools to generate the code for the REST interface.
- Manually coding the REST interface.

Using a specification-first definition, an InterSystems REST service formally consists of the following components:

- A *specification class* (a subclass of %REST.Spec). This class contains the OpenAPI 2.0 specification for the REST service. InterSystems supports several extension attributes that you can use within the specification.
- A *dispatch class* (a subclass of %CSP.REST). This class is responsible for receiving HTTP requests and calling suitable methods in the implementation class.

- An *implementation class* (a subclass of %REST.Impl). This class defines the methods that implement the REST calls.

  The API management tools generate a stub version of the implementation class, which you then expand to include the necessary application logic. (Your logic can of course invoke code outside of this class.)

  The %REST.Impl class provides methods that you can call in order to set HTTP headers, report errors, and so on.

- An InterSystems *web application*, which provides access to the REST service via the InterSystems Web Gateway. The web application is configured to enable REST access and to use the specific dispatch class. The web application also controls access to the REST service.

InterSystems follows a strict naming convention for these components. Given an application name (*appname*), the names of the specification, dispatch, and implementation class are *appname*`.spec`, *appname*`.disp`, and *appname*`.impl`, respectively. The web application is named `/csp/`*appname* by default but you can use a different name for that.

InterSystems supports the specification-first paradigm. You can generate initial code from the specification, and when the specification changes (for example, by acquiring new end points), you can regenerate that code. Later sections provide more details, but for now, note that you should never edit the dispatch class, but can modify the other classes. Also, when you recompile the specification class, the dispatch class is regenerated automatically and the implementation class is updated (preserving your edits).

1.2.1 Manually Coding REST Services

In releases before 2019.2, InterSystems IRIS did not support the specification-first paradigm. A REST service formally consisted only of a dispatch class and a web application. The documentation refers to this way to define REST services as *manually-coded REST services*. The distinction is that a REST service defined by *newer REST service* includes a specification class, and a manually-coded REST service does not. Creating a REST Service Manually describes how to create REST services using the manual coding paradigm. Similarly, *some* of the API management utilities enable you to work with manually-coded REST services.

1.3 Introduction to InterSystems API Management Tools

To help you create REST services more easily, InterSystems provides the following API management tools:

- A REST service named /api/mgmnt, which you can use to discover REST services on the server, generate OpenAPI 2.0 specifications for these REST services, and create, update, or delete REST services on the server.
- The **^%REST** routine, which provides a simple command-line interface that you can use to list, create, and delete REST services.
- The %REST.API class, which you can use to discover REST services on the server, generate OpenAPI 2.0 specifications for these REST services, and create, update, or delete REST services on the server.

You can set up logging for these tools, as described later.

Helpful third-party tools include REST testing tools such as PostMan (https://www.getpostman.com/) and the Swagger editor (https://swagger.io/tools/swagger-editor/download/).

1.4 Overview of Creating REST Services

The recommended way to create REST services in InterSystems products is roughly as follows:

1. Obtain (or write) the OpenAPI 2.0 specification for the service.
2. Use the API management tools to generate the REST service classes and the associated web application. See Creating and Editing REST Services.
3. Modify the implementation class so that the methods contain the suitable business logic. See Modifying the Implementation Class.
4. Optionally modify the specification class. See Modifying the Specification Class. For example, do this if you need to support CORS or use web sessions.
5. If security is required, see Securing REST Services.
6. Using the OpenAPI 2.0 specification for the service, generate documentation as described in Discovering and Documenting REST APIs.

For step 2, another option is to manually create the specification class (pasting the specification into it) and then compile that class; this process generates the dispatch and stub implementation class. That is, it is not strictly necessary to use either the /api/mgmnt service or the **^%REST** routine. This documentation does not discuss this technique further.

1.5 A Closer Look at the REST Service Classes

This section provides a closer look at the specification, dispatch, and implementation classes.

1.5.1 Specification Class

The specification class is meant to define the contract to be followed by the REST service. This class extends %REST.Spec and contains an XData block that contains the OpenAPI 2.0 specification for the REST service. The following shows a partial example:

```
Class petstore.spec Extends %REST.Spec [ ProcedureBlock ]
{

XData OpenAPI [ MimeType = application/json ]
{
{
  "swagger":"2.0",
  "info":{
    "version":"1.0.0",
    "title":"Swagger Petstore",
    "description":"A sample API that uses a petstore as an example to demonstrate features in the
swagger-2.0 specification",
    "termsOfService":"http://swagger.io/terms/",
    "contact":{
      "name":"Swagger API Team"
    },
    "license":{
      "name":"MIT"
    }
  },
...
```

You can modify this class by replacing or editing the specification within the XData block. You can also add class parameters, properties, and methods as needed. Whenever you compile the specification class, the compiler regenerates the dispatch class and updates the implementation class (see How InterSystems Updates the Implementation Class).

1.5.2 Dispatch Class

The dispatch class is directly called when the REST service is invoked. The following shows a partial example:

```
/// Dispatch class defined by RESTSpec in petstore.spec
Class petstore.disp Extends %CSP.REST [ GeneratedBy = petstore.spec.cls, ProcedureBlock ]
{

/// The class containing the RESTSpec which generated this class
Parameter SpecificationClass = "petstore.spec";

/// Default the Content-Type for this application.
Parameter CONTENTTYPE = "application/json";

/// By default convert the input stream to Unicode
Parameter CONVERTINPUTSTREAM = 1;

/// The default response charset is utf-8
Parameter CHARSET = "utf-8";

XData UrlMap [ XMLNamespace = "http://www.intersystems.com/urlmap" ]
{
<Routes>
  <Route Url="/pets" Method="get" Call="findPets" />
  <Route Url="/pets" Method="post" Call="addPet" />
  <Route Url="/pets/:id" Method="get" Call="findPetById" />
  <Route Url="/pets/:id" Method="delete" Call="deletePet" />
</Routes>
}

/// Override %CSP.REST AccessCheck method
ClassMethod AccessCheck(Output pAuthorized As %Boolean) As %Status
{
   ...
}

...
```

Notice that the *SpecificationClass* parameter indicates the name of the associated specification class. The URLMap XData block (the *URL map*) defines the calls within this REST service. It is not necessary for you to have a detailed understanding of this part of the class.

After these items, the class contains the definitions of the methods that are listed in the URL map. Here is one example:

```
ClassMethod deletePet(pid As %String) As %Status
{
    Try {
        If '##class(%REST.Impl).%CheckAccepts("application/json") Do
##class(%REST.Impl).%ReportRESTError(..#HTTP406NOTACCEPTABLE,$$$ERROR($$$RESTBadAccepts)) Quit
        If ($number(pid,"I")="") Do
##class(%REST.Impl).%ReportRESTError(..#HTTP400BADREQUEST,$$$ERROR($$$RESTInvalid,"id",id)) Quit
        Set response=##class(petstore.impl).deletePet(pid)
        Do ##class(petstore.impl).%WriteResponse(response)
    } Catch (ex) {
        Do ##class(%REST.Impl).%ReportRESTError(..#HTTP500INTERNALSERVERERROR,ex.AsStatus())
    }
    Quit $$$OK
}
```

Notice the following points:

- This method invokes a method by the same name in the implementation class (`petstore.impl` in this example). It gets the response from that method and calls **%WriteResponse()** to write the response back to the caller. The **%WriteResponse()** method is an inherited method that is present in all implementation classes, which are all subclasses of %REST.Impl.
- This method does other checking and in case of errors, invokes other methods of %REST.Impl.

Important: Because the dispatch class is a generated class, you should never edit it. InterSystems provides mechanisms for overriding parts of the dispatch class without editing it.

1.5.3 Implementation Class

The implementation class is meant to hold the actual internal implementation of the REST service. You can (and should) edit this class. It initially looks like the following example:

```
/// A sample API that uses a petstore as an example to demonstrate features in the swagger-2.0
specification<br/>
/// Business logic class defined by RESTSpec in petstore.spec<br/>
Class petstore.impl Extends %REST.Impl [ ProcedureBlock ]
{

/// If ExposeServerExceptions is true, then details of internal errors will be exposed.
Parameter ExposeServerExceptions = 0;

/// Returns all pets from the system that the user has access to<br/>
/// The method arguments hold values for:<br/>
///     tags, tags to filter by<br/>
///     limit, maximum number of results to return<br/>
ClassMethod findPets(tags As %ListOfDataTypes(ELEMENTTYPE="%String"), limit As %Integer) As %Stream.Object
{
    //(Place business logic here)
    //Do ..%SetStatusCode(<HTTP_status_code>)
    //Do ..%SetHeader(<name>,<value>)
    //Quit (Place response here) ; response may be a string, stream or dynamic object
}

...
```

The rest of the implementation class contains additional stub methods that look similar to this one. In each case, these stub methods have signatures that obey the contract defined by the specification of the REST service. Note that for the `options` method, InterSystems does not generate a stub method for you to implement. Instead, the class %CSP.REST automatically performs all `options` processing.

1.6 Enabling Logging for API Management Features

To enable logging for the API management features, enter the following in the Terminal:

```
set $namespace="%SYS"
kill ^ISCLOG
set ^%ISCLOG=5
set ^%ISCLOG("Category","apimgmnt")=5
```

Then the system adds entries to the *^ISCLOG* global for any calls to the API management endpoints.

To write the log to a file (for easier readability), enter the following (still within the %SYS namespace):

```
do ##class(%OAuth2.Utils).DisplayLog("filename")
```

Where *filename* is the name of the file to create. The directory must already exist. If the file already exists, it is overwritten.

To stop logging, enter the following (still within the %SYS namespace):

```
set ^%ISCLOG=0
set ^%ISCLOG("Category","apimgmnt")=0
```

1.6.1 Viewing the Log

Once logging for HTTP requests is enabled, the log entries are stored in the *^ISCLOG* global, which is located in the %SYS namespace.

To use the Management Portal to view the log, navigate to **System Explorer** > **Globals** and view the *ISCLOG* global (not *%ISCLOG)*. Make sure you are in the %SYS namespace.

2

Creating and Editing REST Services

There are multiple ways to create and modify REST services within InterSystems IRIS® data platform. The three main methods of doing so are by invoking the `/api/mgmnt/` service, using the ^%REST routine, or using the %REST.API class. These three methods of creating REST services require creating an OpenAPI 2.0 (also called Swagger) description for the REST service to use to generate the service classes. If you are implementing a REST service defined by a third party, they may provide this OpenAPI 2.0 description. See the OpenAPI 2.0 Specification for details about the format of an OpenAPI 2.0 description.

After generating the service classes, see Modifying the Implementation Class and Modifying the Specification Class for further instructions on building a REST service.

2.1 Using the /api/mgmnt/ Service

One of the methods of creating, updating, and deleting REST services involves calling the `/api/mgmnt/` service.

This service also provides options you can use to list and document web services.

2.1.1 Creating REST Services with /api/mgmnt

2.1.1.1 Generating Services Classes with /api/mgmnt/

In the first step, generate the REST service classes, as follows:

1. Create or obtain the OpenAPI 2.0 description of the REST service, in JSON format.
2. Obtain a REST testing tool such as PostMan (https://www.getpostman.com/).
3. In the testing tool, create an HTTP request message as follows:
   - For the HTTP action, select or specify POST.
   - For the URL, specify a URL of the following form, using the <baseURL> for your instance:

     ```
     https://<baseURL>/api/mgmnt/v2/namespace/myapp
     ```

     Where *namespace* is the namespace where you want to create the REST service and *myapp* is the name of the package where you want to create the classes.
   - For the request body, paste the OpenAPI 2.0 description of your web service, in JSON format.
   - Specify the request body type as `JSON (application/json)`

- Provide values for the *IRISUsername* and *IRISPassword* parameters. For *IRISUsername*, specify a user that is a member of the %Developer role and that has read/write access to the given namespace.

4. Send the request message.

   If the call is successful, InterSystems IRIS creates the disp, impl, and spec classes in the given package and namespace.

5. In the testing tool, check the response message. If the request was successful, the response message will look like the following example:

```
{
    "msg": "New application myapp created"
}
```

To complete the basic REST service, create an InterSystems web application and define the implementation (see Modifying the Implementation Class). You can do these steps in either order.

2.1.1.2 Creating the Web Application

In this step, you create a web application that provides access to your REST service. In the Management Portal, complete the following steps:

1. Click **System Administration > Security > Applications > Web Applications**.
2. Click **Create New Web Application**.
3. Specify the following values:
   - **Name** — Name for the web application; this must be unique within this instance of InterSystems IRIS. The most common name is based on the namespace in which the web application runs: `/csp/namespace`
   - **Namespace** — Select the namespace in which you generated the classes.
   - **Enable Application** — Select this check box.
   - **Enable** — Select **REST**.
   - **Dispatch Class** — Type the fully qualified name of the dispatch class. This should always be `package.disp` where *package* is the name of the package that contains the generated classes.

   For information on other options on this page, see Create an Application.
4. Click **Save**.

2.1.2 Updating REST Services with /api/mgmnt/

The InterSystems API management tools enable you to update the generated classes without making changes to your edits in the implementation class. The class is regenerated if necessary, but your edits are preserved.

If the update was successful, InterSystems IRIS regenerates the disp and spec classes in the given package and updates the impl class, preserving edits you made to that class. The response message will look like the following example:

```
{
    "msg": "Application myapp updated"
}
```

2.1.2.1 How InterSystems Updates the Implementation Class

If you previously edited the impl class, InterSystems preserves those edits as follows:

- The implementations of all methods are left as is.

- Any new class members you added are left as is.

However, InterSystems regenerates the description (the `///` comments) of the class and of each generated method. If the signature of any implementation method changes (for example, because the specification has changed), InterSystems updates the signature and adds the following comment to that class method:

```
/// WARNING: This method's signature has changed.
```

2.1.3 Deleting REST Services with /api/mgmnt/

The InterSystems API management tools also enable you to delete a REST service easily. To do so:

1. Using a REST testing tool, create an HTTP request message as follows:
   - For the HTTP action, select or specify DELETE.
   - For the URL, specify a URL of the following form, using the <baseURL> for your instance:

     `http://<baseURL>/api/mgmnt/v2/namespace/myapp`

     Where *localhost* is the name of the server, 52773 is the web server port that InterSystems IRIS is using, *namespace* is the namespace where you want to create the REST service, and *myapp* is the name of the package that contains the REST service classes.
   - Provide values for the *IRISUsername* and *IRISPassword* parameters. For *IRISUsername*, specify a user that is a member of the %Developer role and that has read/write access to the given namespace.
2. Send the request message.

   If the call is successful, InterSystems IRIS deletes the disp and spec classes within the given package and namespace.

   InterSystems IRIS does not, however, delete the impl class.
3. In the testing tool, check the response message. If the request was successful, the response message will look like the following example:

   ```
   {
       "msg": "Application myapp deleted"
   }
   ```
4. Manually delete the implementation class.

   For safety, the /api/mgmnt service does not automatically delete the implementation class, because that class can potentially contain a significant amount of customization.
5. Delete the web application you created previously (if any) for this REST service. To do so:
   a. In the Management Portal, click **System Administration > Security > Applications > Web Applications**.
   b. Click **Delete** in the row that lists the web application.
   c. Click **OK** to confirm the deletion.

2.2 Using the ^%REST Routine

The **^%REST** routine is a simple command-line interface. At any prompt, you can enter the following answers:

^	Causes the routine to skip back to the previous question.
?	Causes the routine to display a message that lists all the current options.
q or quit (not case-sensitive)	Ends the routine.

Also, each question displays, in parentheses, the default answer to that question.

2.2.1 Using ^%REST to Create a Service

The recommended way to create a REST service is to start with the OpenAPI 2.0 specification of the REST service and use that to generate the REST service classes. To use the **^%REST** routine to do this:

1. Obtain the OpenAPI 2.0 specification for the REST service, in JSON format. Either save the specification as a file or make a note of the URL where the specification can be accessed.
2. In the Terminal, change to the namespace where you want to define the REST service.
3. Enter the following command to start the **^%REST** routine:

   ```
   do ^%REST
   ```

4. At the first prompt, enter a name for the REST service. This name is used as the package name for the generated classes; use a valid package name. If you want to use the name `list`, `l`, `quit`, or `q` (in any case variation), enclose the name in double quotes. For example: `"list"`
5. At the next prompt, enter `Y` (not case-sensitive) to confirm that you want to create this service.

   The routine then prompts you for the location of the OpenAPI 2.0 specification to use. Enter either a full pathname or a URL.
6. At the next prompt, enter `Y` (not case-sensitive) to confirm that you want to use this specification.

   The routine creates the disp, impl, and spec classes within the specified package in this namespace. The routine then displays output like the following:

   ```
   -----Creating REST application: myapp-----
   CREATE myapp.spec
   GENERATE myapp.disp
   CREATE myapp.impl
   REST application successfully created.
   ```

   Next the routine asks if you also want to create a web application. You will use this web application to access the REST service.
7. At this point, you can do the following:
   - Enter `Y` (not case-sensitive) to create the web application now.
   - Enter `N` (not case-sensitive) to end the routine.

     You can create the web application separately as described in Creating the Web Application.
8. If you entered `Y`, the routine then prompts you for the name of the web application.

   The name must be unique within this instance of InterSystems IRIS. The default name is based on the namespace in which the web application runs: `/csp/namespace`.

   Enter the name of the web application or press return to accept the default name.

The routine then displays output like the following:

```
-----Deploying REST application: myapp-----
Application myapp deployed to /csp/myapp
```

9. Define the implementation as described in Modifying the Implementation Class.

2.2.2 Using ^%REST to Delete a Service

To use the **^%REST** routine to delete a REST service:

1. In the Terminal, change to the namespace where the REST service can be found.
2. Enter the following command to start the **^%REST** routine:

   ```
   do ^%REST
   ```

3. At the first prompt, enter a name for the REST service.

   If you are not sure of the name of the REST service, enter L (not case-sensitive). The routine lists all the REST services and then prompts you again for the name of the REST service.

4. If the routine finds a REST service with the given name, it displays a prompt like the following:

   ```
   REST application found: petstore
   Do you want to delete the application? Y or N (N):
   ```

5. Enter Y (not case-sensitive) to confirm that you want to delete this service.
6. (Optionally) Manually delete the implementation class.

   For safety, the routine does not automatically delete the implementation class, because that class can potentially contain a significant amount of customization.

2.3 Using the %REST.API Class

This section describes how to use the %REST.API class to create, update, and delete REST services.

2.3.1 Using the %REST.API Class to Create or Update a Service

The recommended way to create a REST service is to start with the OpenAPI 2.0 specification of the REST service and use that to generate the REST service classes. To use the %REST.API class to do this:

1. Obtain the OpenAPI 2.0 specification for the REST service, in JSON format, and save the specification as a file.

   The file must be UTF-8 encoded.

2. In the namespace where you want to define the REST service, use the file to create an instance of %DynamicObject.
3. Then call the **CreateApplication()** method of the %REST.API class. This method has the following signature:

   ```
   classmethod CreateApplication(applicationName As %String,
                                 swagger As %DynamicObject = "",
                                 ByRef features,
                                 Output newApplication As %Boolean,
                                 Output internalError As %Boolean)
                                 as %Status
   ```

   Where:

- *applicationName* is the name of the package where you want to generate the classes.
- *swagger* is the instance of %DynamicObject that represents the OpenAPI 2.0 specification.

  You can also specify this argument as the URL of a specification, the pathname of a file that contains a specification, or as an empty string.
- *features*, which must be passed by reference, is a multidimensional array that holds any additional options:
  - If *features("addPing")* is 1 and if *swagger* is an empty string, then the generated classes include a **ping()** method for testing purposes.
  - If *features("strict")* is 1 (the default), then InterSystems checks all the properties in the specification. If *features("strict")* is 0, then only the properties that are needed for code generation are checked.
- *newApplication*, which is returned as output, is a boolean value that indicates whether the method created a new application (true) or updated an existing application.
- *internalError*, which is returned as output, is a boolean value that indicates whether an internal error occurred.

If the method generates a new application, InterSystems IRIS creates the disp, impl, and spec classes in the given package.

If the method updates an existing application, InterSystems IRIS regenerates the disp and spec classes in the given package and updates the impl class, preserving edits you made to that class.

If the OpenAPI 2.0 specification is invalid, the method does not make any change.

4. Create a web application that access the REST service, as described in Creating and Editing REST Services.
5. Define the implementation as described in Modifying the Implementation Class.

The following shows an example of the first steps:

```
set file="c:/2downloads/petstore.json"
set obj = ##class(%DynamicAbstractObject).%FromJSONFile(file)
do ##class(%REST.API).CreateApplication("petstore",.obj,,.new,.error)
//examine error and decide how to proceed...
...
```

2.3.2 Using the %REST.API Class to Delete a Service

To use the %REST.API class to delete a REST service:

1. In the namespace where the REST service can be found, call the **DeleteApplication()** method of the %REST.API class. This method has the following signature:

   ```
   classmethod DeleteApplication(applicationName As %String) as %Status
   ```

   Where *applicationName* is the name of the package that contains the REST service classes.
2. (Optionally) Manually delete the implementation class.

   For safety, the class method does not automatically delete the implementation class, because that class can potentially contain a significant amount of customization.
3. Delete the web application you created previously (if any) for this REST service. To do so:
   a. In the Management Portal, click **System Administration > Security > Applications > Web Applications**.
   b. Click **Delete** in the row that lists the web application.
   c. Click **OK** to confirm the deletion.

3

Modifying the Implementation Class

This topic discusses how to modify the implementation class for a REST service.

This topic assumes that you have previously generated REST service classes as described in Creating and Editing REST Services.

3.1 Initial Method Definitions

The implementation class initially contains stub methods like the following example:

```
/// Returns all pets from the system that the user has access to<br/>
/// The method arguments hold values for:<br/>
///     tags, tags to filter by<br/>
///     limit, maximum number of results to return<br/>
ClassMethod findPets(tags As %ListOfDataTypes(ELEMENTTYPE="%String"), limit As %Integer) As %Stream.Object
{
    //(Place business logic here)
    //Do ..%SetStatusCode(<HTTP_status_code>)
    //Do ..%SetHeader(<name>,<value>)
    //Quit (Place response here) ; response may be a string, stream or dynamic object
}
```

In each case, these stub methods have signatures that obey the contract defined by the specification of the REST service.

3.2 Implementing the Methods

For each method in the implementation class, edit the method definition (specifically the implementation) as appropriate for the REST call that uses it. Notice that the method is preceded by a comment that is a copy of the description of the corresponding REST call. In the implementation:

- Return the appropriate value.
- Examine the request message. To do so, use the **%CheckAccepts()**, **%GetContentType()**, and **%GetHeader()** methods of the implementation class. All methods mentioned here are inherited from %REST.Impl, the superclass of your implementation class.
- Set the HTTP status code as needed to indicate, for example, whether the resource was available. To do so, use the **%SetStatusCode()** method. For information on HTTP status codes, see http://www.faqs.org/rfcs/rfc2068.html.
- Set the HTTP response header. To do this, use the **%SetHeader()**, **%SetHeaderIfEmpty()**, and **%DeleteHeader()** methods.

- Report errors if needed. To do so, use the **%LogError()** method.

For details on these methods, see the class reference for %REST.Impl.

3.3 Exposing Details of Server Errors

By default, if a REST service encounters an internal error, details of the error are not reported to the client. To change this, add the following to the implementation class and then recompile it:

```
Parameter ExposeServerExceptions = 1;
```

Note that the default **%ReportRESTError()** method checks this parameter. If you override that method (see next heading), you can choose whether your method uses this parameter or not.

3.4 Modifying the Error Response

If you need to format the error response in a non-default way, override the **%ReportRESTError()** method in the implementation class. In your method, use the **%WriteResponse()** method to return the error response.

For details on these methods, see the class reference for %REST.Impl.

4

Modifying the Specification Class

This topic summarizes how and why to modify the specification class for a REST service.

This topic assumes that you have previously generated REST service classes as described in Creating and Editing REST Services.

4.1 Overview

The following table lists reasons for modifying the specification class and briefly summarizes the needed changes:

Reason	Changes
To update or replace the specification	Modify the `OpenAPI` XData block manually or by regenerating the specification class.
Enable REST service to support CORS	Modify the `OpenAPI` XData block manually; also add a class parameter and create a custom dispatch superclass. See Supporting CORS in REST Services.
Enable REST service to support web session	Add a class parameter. See Using Web Sessions with REST.
Specify privileges needed to use endpoints	Modify the `OpenAPI` XData block manually. See Securing REST Services.
Override the default content type, response character set, or input stream handling	Add class parameters. See the next section.
Specify a non-default name for a service method	Modify the `OpenAPI` XData block manually. See Overriding the Name of a Service Method.

Whenever you compile the specification class, the compiler regenerates the dispatch class in the same package and updates the implementation class (see How InterSystems Updates the Implementation Class).

4.2 Overriding the Content Type, Response Character Set, or Input Stream Handling

You can override several key aspects of the REST service simply by adding class parameters to the specification class and recompiling.

- By default, the REST service expects the `application/json` content type. To override this, add the following to the specification class:

  ```
  Parameter CONTENTTYPE = "some-content-type";
  ```

  Where *some-content-type* is a MIME content type.

- By default, the response messages of the REST service are in UTF-8. To override this, add the following to the specification class:

  ```
  Parameter CHARSET = "some-character-set";
  ```

  Where *some-content-type-here* is the name of a character set.

- By default, the REST service converts input character streams to Unicode. To not do this, add the following to the specification class:

  ```
  Parameter CONVERTINPUTSTREAM = 0";
  ```

Then recompile. These changes are then copied to the dispatch class.

4.3 Overriding the Name of a Service Method

By default, the compiler uses the `operationId` of an operation to determine the name of method invoked by the corresponding REST call. You can specify a different name. To do so, add the following to the operation within the `OpenAPI` XData block of the specification class:

```
"x-ISC_ServiceMethod":"alternatename"
```

For example:

```
"/pets":{
  "get":{
    "description":"Returns all pets from the system that the user has access to",
    "operationId":"findPets",
    "x-ISC_ServiceMethod":"ReturnPets",
    "produces":[
      "application/json",
      "application/xml",
      "text/xml",
      "text/html"
    ],
```

Then recompile. The compiler then adds this new method to the dispatch and implementation classes. Be sure to edit the implementation class and provide an implementation for this new method.

4.4 Supporting CORS in REST Services

Cross-Origin Resource Sharing (CORS) allows a script running in another domain to access a service.

Typically, when a browser is running a script from one domain, it allows XMLHttpRequest calls to that same domain but disallows them when they are made to another domain. This browser behavior restricts someone from creating a malicious script that can misuse confidential data. The malicious script could allow the user to access information in another domain using permissions granted to the user, but then, unknown to the user, make other use of confidential information. To avoid this security problem, browsers generally do not allow this kind of cross-domain call.

Without using Cross-Origin Resource Sharing (CORS), a web page with a script accessing REST services typically must be in the same domain as the server providing the REST services. In some environments, it is useful to have the web pages with scripts in a different domain than the servers providing the REST services. CORS enables this arrangement.

The following provides a simplified description of how a browser can handle an XMLHttpRequest with CORS:

1. A script in a web page in domain DomOne contains an XMLHttpRequest to an InterSystems IRIS® data platform REST service that is in domain DomTwo. The XMLHttpRequest has a custom header for CORS.
2. A user views this web page and runs the script. The user's browser detects the XMLHttpRequest to a domain different from the one containing the web page.
3. The user's browser sends a special request to the InterSystems IRIS REST service that indicates the HTTP request method of the XMLHttpRequest and the domain of the originating web page, which is DomOne in this example.
4. If the request is allowed, the response contains the requested information. Otherwise, the response consists only of headers indicating that CORS did not allow the request.

4.4.1 Overview of Enabling a REST Service to Support CORS

By default, InterSystems REST services do not allow the CORS header. You can, however, enable CORS support. There are two parts to enabling support for CORS in a REST service:

- Enabling the REST service to accept the CORS header for some or all HTTP requests. See Accepting the CORS Header.
- Writing code that causes the REST service to examine the CORS requests and decide whether to proceed. For example, you can provide an allow list containing domains that contain only trusted scripts. InterSystems IRIS provides a simple default implementation for documentation purposes; this default implementation allows any CORS request.

Important: The default CORS header processing is not suitable for REST services handling confidential data.

4.4.2 Accepting the CORS Header

To specify that a REST service accepts the CORS header:

1. Modify the specification class to include the *HandleCorsRequest* parameter.

   To enable CORS header processing for all calls, specify the *HandleCorsRequest* parameter as 1:

   ```
   Parameter HandleCorsRequest = 1;
   ```

   Or, to enable CORS header processing for some but not calls, specify the *HandleCorsRequest* parameter as `""` (empty string):

   ```
   Parameter HandleCorsRequest = "";
   ```

2. If you specified *HandleCorsRequest* parameter as `""`, edit the `OpenAPI` XData block in the specification class in order to indicate *which* calls support CORS. Specifically, for the operation objects, add the following property name and value:

   ```
   "x-ISC_CORS":true
   ```

   For example, the OpenAPI XData block might contain this:

   ```
   "post":{
     "description":"Creates a new pet in the store.  Duplicates are allowed",
     "operationId":"addPet",
     "produces":[
       "application/json"
     ],
     ...
   ```

   Add the x-ISC_CORS property as follows:

   ```
   "post":{
     "description":"Creates a new pet in the store.  Duplicates are allowed",
     "operationId":"addPet",
     "x-ISC_CORS":true,
     "produces":[
        "application/json"
     ],
     ...
   ```

3. Compile the specification class. This action regenerates the dispatch class, causing the actual change in behavior. It is not necessary to understand the dispatch class in detail, but notice the following changes:

   - It now contains your value for the *HandleCorsRequest* parameter.
   - The URLMap XData block now includes `Cors="true"` for the `<Route>` element that corresponds to the operation you modified.

If the *HandleCorsRequest* parameter is 0 (the default), then CORS header processing is disabled for all calls. In this case, if the REST service receives a request with the CORS header, the service rejects the request.

Important: An InterSystems IRIS REST service supports the OPTIONS request (the *CORS preflight request*), which is used to determine whether a REST service supports CORS. Users that send such requests should have READ permission on any databases used by the REST service. If not, the service will respond with an HTTP 404 error. In configurations that use delegated authentication, the request will be sent by the authenticated user; assign appropriate permissions in the ZAUTHENTICATE routine. In configurations that do not use delegated authentication, this request is sent unauthenticated and is executed by the CSP-System user; assign appropriate permissions using the management portal.

4.4.3 Defining How to Process the CORS Header

When you enable a REST service to accept the CORS header, by default, the service accepts any CORS request. Your REST service should examine the CORS requests and decide whether to proceed. For example, you can provide an allow list containing domains that contain only trusted scripts. To do this, you need to:

- Create a subclass of %CSP.REST. In this class, implement the **OnHandleCorsRequest()** method as described in the first subsection.
- Modify the specification class and recompile, regenerating the dispatch class.

The net result is that the dispatch class inherits from your custom class instead of from %CSP.REST and thus uses your definition of **OnHandleCorsRequest()**, which overrides the default CORS header processing.

4.4.3.1 Defining OnHandleCorsRequest()

In your subclass of %CSP.REST, define the **OnHandleCorsRequest()** method, which needs to examine the CORS requests and set the response header appropriately.

To define this method, you must be familiar with the details of the CORS protocol (not discussed here).

You also need to know how to examine the requests and set the response headers. For this, it is useful to examine the method that is used by default, the **HandleDefaultCorsRequest()** method of %CSP.REST. This section explains how this method handles the origin, credentials, header, and request method and suggests variations. You can use this information to write your **OnHandleCorsRequest()** method.

The following code gets the origin and uses it to set the response header. One possible variation is to test the origin against an allow list. Then the domain is allowed, set the response header. If not, set the response header to an empty string.

```
#; Get the origin
Set tOrigin=$Get(%request.CgiEnvs("HTTP_ORIGIN"))

#; Allow requested origin
Do ..SetResponseHeaderIfEmpty("Access-Control-Allow-Origin",tOrigin)
```

The following lines specify that the authorization header should be included.

```
#; Set allow credentials to be true
Do ..SetResponseHeaderIfEmpty("Access-Control-Allow-Credentials","true")
```

The following lines get the headers and the request method from the incoming request. Your code should test if the headers and request method are allowed. If they are allowed, use them to set the response headers. If not, set the response header to an empty string.

```
#; Allow requested headers
Set tHeaders=$Get(%request.CgiEnvs("HTTP_ACCESS_CONTROL_REQUEST_HEADERS"))
Do ..SetResponseHeaderIfEmpty("Access-Control-Allow-Headers",tHeaders)

#; Allow requested method
Set tMethod=$Get(%request.CgiEnvs("HTTP_ACCESS_CONTROL_REQUEST_METHOD"))
Do ..SetResponseHeaderIfEmpty("Access-Control-Allow-Method",tMethod)
```

Important: The default CORS header processing is not suitable for REST services handling confidential data.

4.4.3.2 Modifying the Specification Class

After defining your custom subclass of %CSP.REST including an implementation of the **OnHandleCorsRequest()**, do the following:

1. Edit the OpenAPI XData block in the specification class so that the `info` object contains a new property named `x-ISC_DispatchParent`. The value of this property must be the fully qualified name of your custom class.

   For example, suppose that the OpenAPI XData block looks like this:

```
  "swagger":"2.0",
  "info":{
    "version":"1.0.0",
    "title":"Swagger Petstore",
    "description":"A sample API that uses a petstore as an example to demonstrate features in the
 swagger-2.0 specification",
    "termsOfService":"http://swagger.io/terms/",
    "contact":{
      "name":"Swagger API Team"
    },
...
```

Suppose that the custom subclass of %CSP.REST is named `test.MyDispatchClass`. In this case, you would modify the XData block as follows:

```
  "swagger":"2.0",
  "info":{
    "version":"1.0.0",
    "title":"Swagger Petstore",
    "description":"A sample API that uses a petstore as an example to demonstrate features in the
 swagger-2.0 specification",
    "termsOfService":"http://swagger.io/terms/",
    "x-ISC_DispatchParent":"test.MyDispatchClass",
    "contact":{
      "name":"Swagger API Team"
    },
...
```

2. Compile the specification class. This action regenerates the dispatch class. You will notice that the class now extends your custom dispatch superclass. Thus it will use your **OnHandleCorsRequest()** method.

4.5 Using Web Sessions with REST

One of the goals of REST is to be stateless; that is, no knowledge is stored on the server from one REST call to the next. Having a web session preserved across REST calls breaks the stateless paradigm, but there are two reasons why you might want to preserve a web session:

- Minimize connection time — if each REST call creates a new web session, it needs to establish a new session on the server. By preserving a web session, the REST call connects faster.
- Preserve data across REST calls — in some cases, preserving data across REST calls may be necessary to efficiently meet your business requirements.

To enable using a single web session over multiple REST calls, set the *UseSession* parameter to 1 in the specification class. For example:

```
Parameter UseSession As Integer = 1;
```

Then recompile this class.

If *UseSession* is 1, InterSystems IRIS preserves a web session across multiple REST service calls. If the parameter is 0 (the default), InterSystems IRIS uses a new web session for each REST service call.

Note: When you recompile the specification class, the *UseSession* parameter is copied to the dispatch class, which causes the actual change in behavior.

5

Securing REST Services

If your REST service is accessing confidential data, you should use authentication for the service. If you need to provide different levels of access to different users, also specify privileges needed for the endpoints.

This topic assumes that you have previously generated REST service classes as described in Creating and Editing REST Services.

5.1 Setting Up Authentication for REST Services

You can use any of the following forms of authentication with InterSystems IRIS® data platform REST services:

- HTTP authentication headers — This is the recommended form of authentication for REST services.
- Web session authentication — Where the username and password are specified in the URL following a question mark.
- OAuth 2.0 authentication — See the following subsection.

5.1.1 REST Applications and OAuth 2.0

To authenticate a REST application via OAuth 2.0, do all of the following:

- Configure the resource server containing the REST application as an OAuth 2.0 resource server.
- Allow delegated authentication for %Service_WebGateway.
- Make sure that the web application (for the REST application) is configured to use delegated authentication.
- Create a routine named **ZAUTHENTICATE** in the %SYS namespace. InterSystems provides a sample routine, REST.ZAUTHENTICATE.mac, that you can copy and modify. This routine is part of the Samples-Security sample on GitHub (https://github.com/intersystems/Samples-Security). You can download the entire sample as described in Downloading Samples for Use with InterSystems IRIS, but it may be more convenient to simply open the routine on GitHub and copy its contents.

  In your routine, modify the value of *applicationName* and make other changes as needed.

Also see Optionally Defining Delegated Authentication for the Web Client.

Important: If using authentication with HealthShare®, you must use the **ZAUTHENTICATE** routine provided by InterSystems and cannot write your own.

5.2 Specifying Privileges Needed to Use REST Services

To specify privileges needed to execute code or access data, InterSystems technologies use Role-Based Access Control (RBAC). For details, see Authorization: Controlling User Access.

If you need to provide different levels of access to different users, do the following to specify the permissions:

- Modify the specification class to specify the privileges that are needed to use the REST service or specific endpoints in the REST service; then recompile. A privilege is a permission (such as read or write), combined with the name of a resource.

  See the subsection.

- Using the Management Portal:
  - Define the resources that you refer to in the specification class.
  - Define roles that provide sets of privileges. For example, a role could provide read access to an endpoint or write access to a different endpoint. A role can contain multiple sets of privileges.
  - Place users into all the roles needed for their tasks.

Additionally, you can use the SECURITYRESOURCE parameter of the %CSP.REST class to perform authorization.

5.2.1 Specifying Privileges

You can specify a list of privileges for the entire REST service, and you can specify a list of privileges for each endpoint. To do so:

1. To specify the privileges needed to access the service, edit the OpenAPI XData block in the specification class. For the `info` object, add a new property named `x-ISC_RequiredResource` whose value is a comma-separated list of defined resources and their access modes (resource:mode) which are required for access to any endpoint of the REST service.

   The following shows an example:

```
  "swagger":"2.0",
  "info":{
    "version":"1.0.0",
    "title":"Swagger Petstore",
    "description":"A sample API that uses a petstore as an example to demonstrate features in the
swagger-2.0 specification",
    "termsOfService":"http://swagger.io/terms/",
    "x-ISC_RequiredResource":["resource1:read","resource2:read","resource3:read"],
    "contact":{
      "name":"Swagger API Team"
    },
...
```

2. To specify the privileges needed to access a specific endpoint, add the `x-ISC_RequiredResource` property to the operation object that defines that endpoint, as in the following example:

```
      "post":{
        "description":"Creates a new pet in the store.  Duplicates are allowed",
        "operationId":"addPet",
        "x-ISC_RequiredResource":["resource1:read","resource2:read","resource3:read"],
        "produces":[
          "application/json"
        ],
        ...
```

3. Compile the specification class. This action regenerates the dispatch class.

5.2.2 Using the SECURITYRESOURCE Parameter

As an additional authorization tool, dispatch classes that subclass %CSP.REST have a *SECURITYRESOURCE* parameter. The value of *SECURITYRESOURCE* is either a resource and its permission or simply the resource (in which case the relevant permission is Use). The system checks if a user has the required permission on the resource associated with *SECURITYRESOURCE*.

Note: If the dispatch class specifies a value for *SECURITYRESOURCE* and the CSPSystem user is not sufficiently privileged, then this may result in unexpected HTTP error codes for failed login attempts. To prevent this from occurring, InterSystems recommends that you give permissions on the specified resource to the CSPSystem user.

6

Listing and Documenting REST APIs

This topic discusses how to discover the REST services that are available on an instance and how to generate documentation for REST services.

6.1 Using the /api/mgmnt Service to Discover REST Services

The /api/mgmnt service includes calls you can use to discover REST service classes and REST-enabled web applications.

6.1.1 Discovering REST Services

To use the /api/mgmnt service to discover the REST services that are available on an instance, use the following REST call:

- For the HTTP action, select or specify GET.
- For the URL, specify a URL of the following form, using the <baseURL> for your instance:

  `http://<baseURL>/api/mgmnt/v2/`

  Or, if you want to examine only one namespace:

  `http://<baseURL>/api/mgmnt/v2/:namespace`

  Where *namespace* is the namespace you want to examine.

(Note that these calls ignore manually-coded REST services. To discover manually-coded REST applications, use the calls GET /api/mgmnt/ and GET /api/mgmnt/:v1/:namespace/restapps.)

If the call is successful, InterSystems IRIS returns an array that lists the REST services, in JSON format. For example:

```
[
  {
    "name": "%Api.Mgmnt.v2",
    "webApplications": "/api/mgmnt",
    "dispatchClass": "%Api.Mgmnt.v2.disp",
    "namespace": "%SYS",
    "swaggerSpec": "/api/mgmnt/v2/%25SYS/%Api.Mgmnt.v2"
  },
  {
    "name": "myapp",
    "webApplications": "/api/myapp",
    "dispatchClass": "myapp.disp",
    "namespace": "USER",
    "swaggerSpec": "/api/mgmnt/v2/USER/myapp"
  }
]
```

6.1.2 Discovering REST-Enabled Web Applications

To use the /api/mgmnt service to discover the REST-enabled web applications that are available on an instance, use the following REST call:

- For the HTTP action, select or specify GET.
- For the URL, specify a URL of the following form, using the <baseURL> for your instance:

  `http://<baseURL>/api/mgmnt`

  Or, if you want to examine only one namespace:

  `http://<baseURL>/api/mgmnt/v1/:namespace/restapps`

  Where *namespace* is the namespace you want to examine.

See the reference sections for GET /api/mgmnt/ and GET /api/mgmnt/:v1/:namespace/restapps.

6.2 Using the %REST.API Class to Discover REST Services

The %REST.API class provides methods you can use to discover REST service classes and REST-enabled web applications.

6.2.1 Discovering REST Service Classes

To use the %REST.API class to discover the REST services that are available on an instance, use the following methods of that class:

GetAllRESTApps()

```
GetAllRESTApps(Output appList As %ListOfObjects) as %Status
```

Returns, as output, a list of the REST services on this server. The output argument *applist* is an instance of %ListOfObjects, and each item in the list is an instance of %REST.Application that contains information about the REST service. This includes any REST services that do not have an associated web application. This method ignores any manually-coded REST services.

GetRESTApps()

```
GetRESTApps(namespace as %String,
            Output appList As %ListOfObjects) as %Status
```

Returns, as output, a list of the REST services in the namespace indicated by *namepace*. See **GetAllWebRESTApps()**. See **GetAllRESTApps()**.

6.2.2 Discovering REST-Enabled Web Applications

To use the %REST.API class to discover the REST-enabled web applications that are available on an instance, use the following methods of that class:

GetAllWebRESTApps()

```
GetAllWebRESTApps(Output appList As %ListOfObjects) as %Status
```

Returns, as output, a list of the REST-enabled web applications on this server. The output argument *applist* is an instance of %ListOfObjects, and each item in the list is an instance of %REST.Application that contains information about web application.

GetWebRESTApps()

```
GetWebRESTApps(namespace as %String,
               Output appList As %ListOfObjects) as %Status
```

Returns, as output, a list of the REST-enabled web applications in the namespace indicated by *namepace*. See **GetAllWebRESTApps()**.

6.3 Providing Documentation for a REST Service

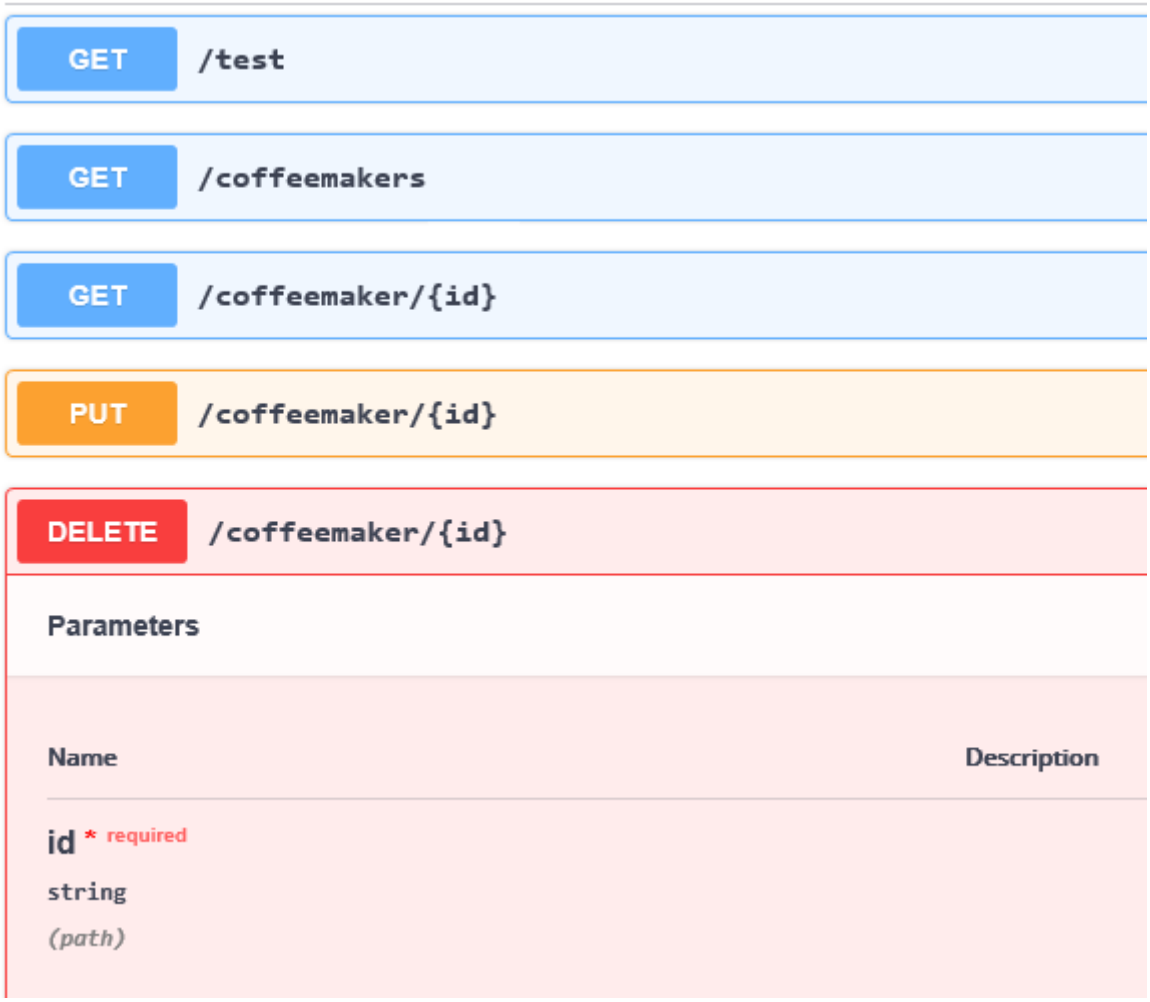
It is useful to document any API so that developers can easily use the API. In the case of a REST API that follows the OpenAPI 2.0 specification, you can use the Swagger open-source framework to provide interactive documentation for your API, based upon the contents of the specification.

One option is to use Swagger UI and provide a hosted copy of the documentation. For a demo:

1. Go to https://swagger.io/tools/swagger-ui/
2. Click **Live Demo**.
3. In the box at the top of the page, enter the URL of the OpenAPI 2.0 specification for the REST service, in JSON format.

   For example, use the GET /api/mgmnt/v2/:namespace/:application call on the InterSystems IRIS server.
4. Click **Explore**.

The lower part of the page then displays the documentation as shown in the following example:



Here you can view details about each call, try test calls and see the responses. For more details, see the Swagger web site.

Other third-party tools enable you to generate static HTML. InterSystems has no specific recommendations for this.

/api/mgmnt/ API Endpoints

This reference lists the endpoints in the /api/mgmnt/ service, all of which apply to newer REST services. The following table summarizes the endpoints and indicates whether they also apply to manually-coded REST services.

Endpoint	Summary	Applies to NEWER REST Services?	Applies to Manually-Coded REST Services?
DELETE /api/mgmnt/v2/:ns/:app	Deletes the REST service	*YES*	no
GET /api/mgmnt/	Lists the REST-enabled web applications on this server	*YES*	*YES*
GET /api/mgmnt/v1/:ns/restapps	Lists the REST-enabled web applications in the namespace	*YES*	*YES*
GET /api/mgmnt/v1/:ns/spec/:app	Returns the OpenAPI 2.0 specification for the REST service	no	*YES*
GET /api/mgmnt/v2/	Lists the REST services on this server (including any that do not have an associated web application)	*YES*	no
GET /api/mgmnt/v2/:ns	Lists the REST services in the namespace (including any that do not have an associated web application)	*YES*	no
GET /api/mgmnt/v2/:ns/:app	Returns the OpenAPI 2.0 specification for the REST service	*YES*	*YES*

Here *ns* is a namespace, and *app* is the name of the package that contains the REST service classes.

DELETE /api/mgmnt/v2/:namespace/:application/

Deletes the classes for the given REST application. Note that this call looks for a newer REST service. It ignores any manually-coded REST services.

URL Parameters

namespace	*Required*. Namespace name. This parameter is not case-sensitive.
application	*Required*. Fully qualified name of the package that contains the spec, impl, and disp classes. This parameter is not case-sensitive.

Permissions

To use this endpoint, you must be a member of the %Developer role and must have read/write access to the given namespace.

Example Request

- *Request Method:*

  ```
  DELETE
  ```

- *Request URL:*

  ```
  http://localhost:52773/api/mgmnt/v2/user/myapp
  ```

Response

There is no response to this call.

GET /api/mgmnt/

Returns an array that contains information about REST-enabled web applications in all namespaces.

URL Parameters

None.

Permissions

To use this endpoint, you must have read access to the given namespace. If no namespace is specified, or if the specified namespace is %SYS, you must have read access to the default namespace (USER). Note that you can set the default namespace to a different namespace; to do so, set the global node `^%SYS("REST","UserNamespace")` equal to the desired namespace.

Example Request

- *Request Method:*

  `GET`

- *Request URL:*

  `http://localhost:52773/api/mgmnt/`

Response

The response is a JSON array; each object in the array represents a REST service on this server. Specifically, this call retrieves information about all REST-enabled web applications configured on this server. It finds both newer and manually-coded REST services. If there are REST service classes (newer or manually-coded) that have no associated REST-enabled web application, those are not included in this response.

A given object has the following properties:

- `name` — Name of the REST-enabled web application.
- `dispatchClass` — Name of the dispatch class of the REST service. Specifically this is the class indicated by the **Dispatch Class** configuration option of the web application.
- `namespace` — Namespace in which the dispatch class is defined.
- `resource` — Name of the InterSystems IRIS® data platform resource needed to use this REST service.
- `swaggerSpec` — Endpoint where you can obtain the OpenAPI 2.0 specification for this REST service.
- `enabled` — Specifies whether the REST service is enabled or not.

The following shows a sample response:

```
[
    {
        "name": "/api/atelier",
        "dispatchClass": "%Api.Atelier",
        "namespace": "%SYS",
        "resource": "%Development",
        "swaggerSpec": "/api/mgmnt/v1/%25SYS/spec/api/atelier",
        "enabled": true
    },
    {
        "name": "/api/deepsee",
        "dispatchClass": "%Api.DeepSee",
        "namespace": "%SYS",
        "resource": "",
        "swaggerSpec": "/api/mgmnt/v1/%25SYS/spec/api/deepsee",
        "enabled": true
```

```
    },
    {
        "name": "/api/docdb",
        "dispatchClass": "%Api.DocDB",
        "namespace": "%SYS",
        "resource": "",
        "swaggerSpec": "/api/mgmnt/v1/%25SYS/spec/api/docdb",
        "enabled": true
    },
    {
        "name": "/api/iknow",
        "dispatchClass": "%Api.iKnow",
        "namespace": "%SYS",
        "resource": "",
        "swaggerSpec": "/api/mgmnt/v1/%25SYS/spec/api/iknow",
        "enabled": true
    },
    {
        "name": "/api/mgmnt",
        "dispatchClass": "%Api.Mgmnt.v2.disp",
        "namespace": "%SYS",
        "resource": "",
        "swaggerSpec": "/api/mgmnt/v1/%25SYS/spec/api/mgmnt",
        "enabled": true
    },
    {
        "name": "/api/uima",
        "dispatchClass": "%Api.UIMA",
        "namespace": "%SYS",
        "resource": "",
        "swaggerSpec": "/api/mgmnt/v1/%25SYS/spec/api/uima",
        "enabled": true
    },
    {
        "name": "/webapp/simple2",
        "dispatchClass": "simple2.disp",
        "namespace": "USER",
        "resource": "",
        "swaggerSpec": "/api/mgmnt/v1/USER/spec/webapp/simple2",
        "enabled": true
    }
]
```


See Also

- GET /api/mgmnt/v2/

GET /api/mgmnt/v1/:namespace/restapps

Returns an array that contains information about REST-enabled web applications in the given namespace.

URL Parameters

None.

Permissions

To use this endpoint, you must have read access to the given namespace. If no namespace is specified, or if the specified namespace is %SYS, you must have read access to the default namespace (USER). Note that you can set the default namespace to a different namespace; to do so, set the global node `^%SYS("REST","UserNamespace")` equal to the desired namespace.

Example Request

- *Request Method:*

  `GET`

- *Request URL:*

  `http://localhost:52773/api/mgmnt/v1/user/restapps`

Response

The response is a JSON array; each object in the array represents a REST-enabled web application. For details, see GET /api/mgmnt/.

See Also

- GET /api/mgmnt/v2/:namespace

GET /api/mgmnt/v1/:namespace/spec/:application/

Returns the OpenAPI 2.0 specification for the given REST service, which must be a manually-coded REST service.

URL Parameters

namespace	*Required*. Namespace name. This parameter is not case-sensitive.
application	*Required*. Fully qualified name of the package that contains the REST service class.

Permissions

To use this endpoint, you must have read access to the given namespace. If no namespace is specified, or if the specified namespace is %SYS, you must have read access to the default namespace (USER). Note that you can set the default namespace to a different namespace; to do so, set the global node `^%SYS("REST","UserNamespace")` equal to the desired namespace.

You must also have read access to the database that contains the dispatch class. The dispatch class namespace and the endpoint namespace need to be the same (except in the case of a dispatch class in a package starting with %, which is available to all namespaces).

Example Request

- *Request Method:*

  `GET`

- *Request URL:*

  `http://localhost:52773/api/mgmnt/v1/user/spec/myapp`

Response

This call returns a Swagger (OpenAPI 2.0) specification as documented at https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md.

GET /api/mgmnt/v2/

Returns an array that contains information about the newer REST services on the server (including any that do not have an associated web application). This call ignores any manually-coded REST services.

URL Parameters

None.

Permissions

To use this endpoint, you must have read access to the given namespace. If no namespace is specified, or if the specified namespace is %SYS, you must have read access to the default namespace (USER). Note that you can set the default namespace to a different namespace; to do so, set the global node `^%SYS("REST","UserNamespace")` equal to the desired namespace.

Example Request

- *Request Method:*

  `GET`

- *Request URL:*

  `http://localhost:52773/api/mgmnt/v2/`

Response

The response is a JSON array; each object in the array represents a REST service. A given object has the following properties:

- `name` — Name of the REST service.
- `webApplications` — Name of the web application that provides access to the REST service.
- `dispatchClass` — Name of the dispatch class of the REST service.
- `namespace` — Namespace in which the dispatch class and other classes are defined.
- `swaggerSpec` — Endpoint where you can obtain the OpenAPI 2.0 specification for this REST service.

The following shows an example response:

```
[
  {
    "name": "%Api.Mgmnt.v2",
    "webApplications": "/api/mgmnt",
    "dispatchClass": "%Api.Mgmnt.v2.disp",
    "namespace": "%SYS",
    "swaggerSpec": "/api/mgmnt/v2/%25SYS/%Api.Mgmnt.v2"
  },
  {
    "name": "myapp",
    "webApplications": "/api/myapp",
    "dispatchClass": "myapp.disp",
    "namespace": "USER",
    "swaggerSpec": "/api/mgmnt/v2/USER/myapp"
  }
]
```

GET /api/mgmnt/v2/:namespace/

Returns an array that contains information about the newer REST services in the given namespace (including any REST services that do not have an associated web application). This call ignores any manually-coded REST services.

URL Parameters

namespace	*Required*. Namespace name. This parameter is not case-sensitive.

Permissions

To use this endpoint, you must have read access to the given namespace. If no namespace is specified, or if the specified namespace is %SYS, you must have read access to the default namespace (USER). Note that you can set the default namespace to a different namespace; to do so, set the global node `^%SYS("REST","UserNamespace")` equal to the desired namespace.

Example Request

- *Request Method:*

  `GET`

- *Request URL:*

  `http://localhost:52773/api/mgmnt/v2/%25sys/`

Response

The response is a JSON array; each object in the array represents a REST service. For details, see GET /api/mgmnt/v2/.

The following shows an example response:

```
[
  {
    "name": "%Api.Mgmnt.v2",
    "webApplications": "/api/mgmnt",
    "dispatchClass": "%Api.Mgmnt.v2.disp",
    "namespace": "%SYS",
    "swaggerSpec": "/api/mgmnt/v2/%25SYS/%Api.Mgmnt.v2"
  }
]
```

GET /api/mgmnt/v2/:namespace/:application/

Returns the OpenAPI 2.0 specification for the given REST service. The REST service can be either a newer REST service or a manually-coded REST service.

URL Parameters

namespace	*Required*. Namespace name. This parameter is not case-sensitive.
application	*Required*. Fully qualified name of the package that contains the spec, impl, and disp classes. This parameter is not case-sensitive.

Permissions

To use this endpoint, you must have read access to the given namespace. If no namespace is specified, or if the specified namespace is %SYS, you must have read access to the default namespace (USER). Note that you can set the default namespace to a different namespace; to do so, set the global node `^%SYS("REST","UserNamespace")` equal to the desired namespace.

You must also have read access to the database that contains the dispatch class. The dispatch class namespace and the endpoint namespace need to be the same (except in the case of a dispatch class in a package starting with %, which is available to all namespaces).

Example Request

- *Request Method:*

  `GET`

- *Request URL:*

  `http://localhost:52773/api/mgmnt/v2/user/myapp`

Response

This call returns a Swagger (OpenAPI 2.0) specification as documented at https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md. In the returned specification:

- `host` is always set to the *server*`:`*port* of the server that you called.
- `basePath` is set to the first web application that dispatches to this REST application. If no web application dispatches to this REST application, then `basePath` is returned as in the .spec class.

POST /api/mgmnt/v2/:namespace/:application

Given a Swagger (OpenAPI 2.0) specification, this call generates the scaffolding for the REST application.

URL Parameters

nam/espace	*Required*. Namespace name. This parameter is not case-sensitive.
application	*Required*. Fully qualified name of the package that contains the spec, impl, and disp classes. This parameter is not case-sensitive.

Request Body

The request body must be a Swagger (OpenAPI 2.0) specification in JSON format.

Permissions

To use this endpoint, you must be a member of the %Developer role and must have read/write access to the given namespace.

Example Request

- *Request Method:*

  `POST`

- *Request URL:*

  `http://localhost:52773/api/mgmnt/v2/user/myapp`

- *Request Body:*

  A Swagger (OpenAPI 2.0) specification in JSON format.

A

OpenAPI Properties in Use

This page lists the properties of the OpenAPI 2.0 specification that the API management tools use when generating the REST service classes. Properties not listed here are ignored. There are several extension properties; these have names that start with `x-ISC`.

A.1 Swagger

- `basePath`
- `consumes`
- `host`
- `produces`
- `definitions` (note that the API management tools do not use any properties of the Schema object when generating code)
- `parameters` (for details, see Parameter Object)
- `paths` (for details, see Path Item Object)
- `info` (for details, see Info Object)
- `swagger` (must be `"2.0"`)

For details on these properties, see https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#swagger-object.

A.2 Info Object

- `title`
- `description`
- `x-ISC_RequiredResource` (a comma-separated list of defined resources and their access modes (*resource*:*mode*) that are required for access to any endpoint of the REST service)
- `version`

For details on the standard properties, see https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#info-object.

A.3 Path Item Object

- `$ref`
- `get`, `put`, and so on (all methods listed in OpenAPI 2.0 specification are supported)

  Note that for the `options` method, InterSystems does not generate a stub method for you to implement. Instead, the class %CSP.REST automatically performs all `options` processing.
- `parameters` (for details, see Parameter Object)

For details on these properties, see https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#pathItemObject.

A.4 Operation Object

- `operationId`
- `summary`
- `description`
- `consumes`
- `produces`
- `parameters` (for details, see Parameter Object)
- `x-ISC_CORS` (a flag to indicate that CORS requests for this endpoint/method combination should be supported)
- `x-ISC_RequiredResource` (a comma-separated list of defined resources and their access modes (*resource*:*mode*) that are required for access to this endpoint of the REST service)
- `x-ISC_ServiceMethod` (name of the class method called on the back end to service this operation; default is `operationId`, which is normally suitable)
- `responses` (note that within the response object, `status` may be HTTP status code or `"default"`)

For details on the standard properties, see https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#operationObject.

A.5 Parameter Object

- `name`
- `in`
- `description`
- `required`

- `$ref`
- `type` (cannot be `"formData"`; other types are permitted)
- `format`
- `allowEmptyValue`
- `maxLength`
- `minLength`
- `pattern`
- `maximum`
- `minimum`
- `exclusiveMaximum`
- `exclusiveMinimum`
- `multipleOf`
- `collectionFormat`
- `minItems`
- `maxItems`
- `uniqueItems`
- `items` (for details, see Items Object)

For details on these properties, see https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#parameter-object.

A.6 Items Object

- `type`
- `format`
- `allowEmptyValue`
- `maxLength`
- `minLength`
- `pattern`
- `maximum`
- `minimum`
- `exclusiveMaximum`
- `exclusiveMinimum`
- `multipleOf`
- `collectionFormat`
- `minItems`

- `maxItems`
- `uniqueItems`

For details on these properties, see https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#items-object.

B

Creating a REST Service Manually

This page describes how to manually create an InterSystems IRIS® data platform REST service by subclassing the %CSP.REST class; this procedure creates a manually-coded REST service that does not work with all the API management tools.

Also see Setting Up Authentication for REST Services in Securing REST Services.

B.1 Basics of Creating a REST Service Manually

To define a REST service manually, do the following:

- Create a *REST service class* — subclass of %CSP.REST. In your subclass:
  - Define a URL map that specifies the InterSystems IRIS method that is executed for a REST URL and HTTP method.
  - Optionally specify the *UseSession* parameter. This parameter controls whether each REST call is executed under its own web session or shares a single session with other REST calls.
  - Optionally, override error handling methods.

  If you want to separate the implementation code from the dispatch code, you can define the methods implementing the REST services in separate class and invoke those methods from the URL map.
- Define a web application that uses the REST service class as its dispatch class.

  To define the web application and its security, go to the **Web Application** page (click **System Administration** > **Security** > **Applications** > **Web Applications**).

  When you define the web application, you set the **Dispatch Class** to the name of your REST service class.

  Also, specify the name of the application as the first part of the URL for the REST calls. An example name is `/csp/mynamespace` or `/csp/myapp`, but you can specify any text that is allowed in a URL.

You can define more than one REST service class in a namespace. Each REST service class that has its own entry point must have its own web application.

B.2 Creating the URL Map

In the REST service class, define an XData block named UrlMap that associates the REST call with the method that implements the service. It can either directly send the call to a method based on the contents of the URL or it can forward the call to another REST service class based on the URL. If the web application is handling a small number of related services, you can send the call directly to the method that implements it. However, if the web application is handling a large number of disparate services, you can define separate REST service classes, each of which handles a set of related services. Then configure the web application to use a central REST service class that forwards the REST calls to other REST service classes as appropriate.

If the subclass of %CSP.REST is sending the call directly to the methods, the URLMap contains a `<Routes>` definition that contains a series of `<Route>` elements. Each `<Route>` element specifies a class method to be called for the specified URL and HTTP operation. Typically REST uses the GET, POST, PUT, or DELETE operations, but you can specify any HTTP operation. The URL can optionally include parameters that are specified as part of the REST URL and passed to the specified method as parameters.

If the subclass of %CSP.REST is forwarding the calls to other subclasses of %CSP.REST, the UrlMap contains a `<Routes>` definition that contains a series of `<Map>` elements. The `<Map>` element forwards all calls with the specified prefix to another REST service class, which will then implement the behavior. It can implement the behavior by sending the call directly to a method or by forwarding it to another subclass.

Important: InterSystems IRIS compares the incoming REST URL with the `URL` property of each `<Route>` and the `Prefix` property of each `<Map>`, starting with the first item in the URL map, and stopping at the first possible match that uses the same HTTP request method. Thus the order of the elements in the `<Routes>` is significant. If an incoming URL could match multiple elements of the URL map, InterSystems IRIS uses the first matching element and ignores any subsequent possible matches.

B.2.1 URLMap with <Route> Elements

InterSystems IRIS compares the incoming URL and the HTTP request method to each `<Route>` element in the URL map. It calls the method specified in the first matching `<Route>` element. The `<Route>` element has three parts:

- `Url` — specifies the format of the last part of the REST URL to call the REST service. The Url consists of text elements and parameters prefaced by : (colon).
- `Method` — specifies the HTTP request method for the REST call: typically these are GET, POST, PUT, or DELETE, but any HTTP request method can be used. You should choose the request method that is appropriate for the function being performed by the service, but the %CSP.REST class does not perform any special handling of the different method. You should specify the HTTP request method in all uppercase letters.
- `Call` — specifies the class method to call to perform the REST service. By default, this class method is defined in your REST service class, but you can explicitly specify any class method.

For example, consider the following `<Route>`:

```
<Route Url="/echo" Method="POST" Call="Echo" Cors="false" />
```

This specifies that the REST call will end with `/echo` and use the POST method. It will call the Echo class method in the REST.DocServer class that defines the REST service. The **Cors** property is optional; see Modifying a REST Service to Use CORS for details.

The complete REST URL consists of the following pieces:

- The base URL of your InterSystems IRIS instance.

- The name of the web application as defined on the **Web Application** page (click **System Administration** > **Security** > **Applications** > **Web Applications**). (For example, `/csp/samples/docserver`)
- The `Url` property of the `<Route>` element. If a segment of the `Url` property is preceded by a : (colon), it represents a parameter. A parameter will match any value in that URL segment. This value is passed to the method as a parameter.

For the preceding example, the complete REST call as shown by a TCP tracing utility is:

```
POST /csp/samples/docserver/echo HTTP/1.1
Host: localhost:52773
```

If you want to separate the code implementing the REST services from the %CSP.REST dispatch code, you can define the methods implementing the REST services in another class and specify the class and method in the Call element.

B.2.1.1 Specifying Parameters

The following `<Route>` definition defines two parameters, namespace and class, in the URL:

```
<Route Url="/class/:namespace/:classname" Method="GET" Call="GetClass" />
```

A REST call URL starts with `/csp/samples/docserver/class/` and the next two elements of the URL specify the two parameters. The **GetClass()** method uses these parameters as the namespace and the class name that you are querying. For example, consider this REST call:

```
http://localhost:52773/csp/samples/docserver/class/samples/Cinema.Review
```

This REST call invokes the **GetClass()** method and passes the strings `"samples"` and `"Cinema.Review"` as the parameter values. The **GetClass()** method has the following signature:

```
/// This method returns the class text for the named class
ClassMethod GetClass(pNamespace As %String,
                     pClassname As %String) As %Status
{
```

B.2.1.2 Specifying Multiple Routes for a Single URL

For a given URL, you can support different HTTP request methods. You can do either by defining separate ObjectScript methods for each HTTP request, or by using a single ObjectScript method that examines the request.

The following example uses different methods for each HTTP request method for a single URL:

```
<Route Url="/request" Method="GET" Call="GetRequest" />
<Route Url="/request" Method="POST" Call="PostRequest" />
```

With these routes, if the URL `/csp/samples/docserver/request` is called with an HTTP GET method, the GetRequest() method is invoked. If it is called with an HTTP POST method, the PostRequest() method is invoked.

In contrast, you could use the following `<Route>` definitions:

```
<Route Url="/request" Method="GET" Call="Request" />
<Route Url="/request" Method="POST" Call="Request" />
```

In this case, the **Request()** method handles the call for either a GET or a POST operation. The method examines the *%request* object, which is an instance of %CSP.Request. In this object, the *URL* property contains the text of the URL.

B.2.1.3 Regular Expressions in the Route Map

You can use regular expressions within the route map. InterSystems suggests that you do so only if there is no other way to define the REST service to meet your needs. This section provides the details. (For information on regular expressions in ObjectScript, see Regular Expressions.)

Internally, the :*parameter-name* syntax for defining parameters in the URL is implemented using regular expressions. Each segment specified as :*parameter-name* is converted to a regular expression that contains a repeating matching group, specifically to the `([^/]+)` regular expression. This syntax matches any string (of non-zero length), as long as that string does not include the / (slash) character. So the **GetClass()** sample, which is `Url="/class/:namespace/:classname"`, is equivalent to:

```
<Route Url="/class/([^/]+)/([^/]+)" Method="GET" Call="GetClass" />
```

where there are two matching groups that specify two parameters.

In most cases this format provides enough flexibility to specify the REST URL, but advanced users can use the regular expression format directly in the route definition. The URL must match the regular expression, and each matching group, which is specified by a pair of parentheses, defines a parameter to be passed to the method.

For example, consider the following route map:

```
<Routes>
<Route Url="/Move/:direction" Method="GET" Call="Move" />
<Route Url="/Move2/(east|west|north|south)" Method="GET" Call="Move" />
</Routes>
```

For the first route, the parameter can have any value. No matter what value the parameter has, the **Move()** method is called. For the second route, the parameter must be one of `east` `west` `north` or `south`; if you call the second route with a parameter value other than those, the **Move()** method is not called, and REST service returns a 404 error because the resource cannot be found.

This simple example is meant only to demonstrate the difference between the usual parameter syntax and a regular expression. In the case discussed here, there is no need for a regular expression because the **Move()** method can (and should) check the value of the parameter and respond appropriately. In the following cases, however, a regular expression is helpful:

- *If a parameter is optional.* In this case, use the regular expression `([^/]*)` instead of the :*parameter-name* syntax. For example:

  ```
  <Route Url="/Test3/([^/]*)" Method="GET" Call="Test"/>
  ```

  Of course, the method being called must also be able to handle having a null value for the parameter.

- *If the parameter is the last parameter and its value can include a slash.* In this case, if the parameter is required, use the regular expression `((?s).+)` instead of the :*parameter-name* syntax. For example:

  ```
  <Route Url="/Test4/((?s).+)" Method="GET" Call="Test"/>
  ```

  Or, if this parameter is optional, use the regular expression `((?s).*)` instead of the :*parameter-name* syntax. For example:

  ```
  <Route Url="/Test5/((?s).*)" Method="GET" Call="Test"/>
  ```

B.2.2 URLMap with <Map> Elements

InterSystems IRIS compares the incoming URL to the prefix in each `<Map>` element in the URL map. It forwards the incoming REST call to the REST service class specified in the first matching `<Map>` element. That class processes the remainder of the URL, typically calling the method that implements the service. The `<Map>` element has two attributes:

- `Prefix` — specifies the segment of the URL to match. The incoming URL typically has other segments after the matching segment.
- `Forward` — specifies another REST service class that will process the URL segments that follow the matching segment.

Consider the following URLMap that contains three <Map> elements.

```
XData UrlMap
{
  <Routes>
    <Map Prefix="/coffee/sales" Forward="MyLib.coffee.SalesREST"/>
    <Map Prefix="/coffee/repairs" Forward="MyLib.coffee.RepairsREST"/>
    <Map Prefix="/coffee" Forward="MyLib.coffee.MiscREST"/>
  </Routes>
}
```

This UrlMap forwards the REST call to one of three REST service classes: MyLib.coffee.SalesREST, MyLib.coffee.RepairsREST, or MyLib.coffee.MiscREST.

The complete REST URL to call one of these REST services consists of the following pieces:

- The base URL
- Name of the web application as defined on the **Web Application** page (click **System Administration** > **Security** > **Applications** > **Web Applications**). For example, the web application for these REST calls could be named /coffeeRESTSvr
- The Prefix for a <Map> element.
- The remainder of the REST URL. This is the URL that will be processed by the REST service class that receives the forwarded REST request.

For example, the following REST call:

```
http://localhost:52773/coffeeRESTSvr/coffee/sales/reports/id/875
```

matches the first <Map> with the prefix /coffee/sales and forwards the REST call to the MyLib.coffee.SalesREST class. That class will look for a match for the remainder of the URL, "/reports/id/875".

As another example, the following REST call:

```
http://localhost:52773/coffeeRESTSvr/coffee/inventory/machinetype/drip
```

matches the third <Map> with the prefix /coffee and forwards the REST call to the MyLib.coffee.MiscREST class. That class will look for a match for the remainder of the URL, "/inventory/machinetype/drip".

Note: In this URLMap example, if the <Map> with the prefix="/coffee" was the first map, all REST calls with /coffee would be forwarded to the MyLib.coffee.MiscREST class even if they matched one of the following <Map> elements. The order of the <Map> elements in <Routes> is significant.

B.3 Specifying the Data Format

You can define your REST service to handle data in different formats, such as JSON, XML, text, or CSV. A REST call can specify the form that it expects data it is sending by specifying a ContentType element in the HTTP request and can request the return data format by specifying an Accept element in the HTTP request.

In the DocServer sample, the **GetNamespaces()** method checks if the REST call requested JSON data with the following:

```
If $Get(%request.CgiEnvs("HTTP_ACCEPT"))="application/json"
```

B.4 Localizing a REST Service

Any string value returned by a REST service can be localized, so that the server stores multiple versions of the strings in different languages. Then when the service receives an HTTP request that includes the HTTP `Accept-Language` header, the service responds with the appropriate version of the string.

To localize a REST service:

1. Within your implementation code, rather than including a hardcoded literal string, use an instance of the `$$$Text` macro, providing values for the macro arguments as follows:
   - The default string
   - (Optional) The domain to which this string belongs (localization is easier to manage when the strings are grouped into domains)
   - (Optional) The language code of the default string

   For example, instead of this:

   ```
   set returnvalue="Hello world"
   ```

   Include this:

   ```
   set returnvalue=$$$TEXT("Hello world","sampledomain","en-us")
   ```

2. If you omit the domain argument to `$$$Text` macro, also include the *DOMAIN* class parameter within the REST service class. For example:

   ```
   Parameter DOMAIN = "sampledomain"
   ```

3. Compile the code. When you do so, the compiler generates entries in the message dictionary for each unique instance of the `$$$Text` macro.

   The message dictionary is a global and so can be easily viewed (for example) in the Management Portal. There are class methods to help with common tasks.

4. When development is complete, export the message dictionary for that domain or for all domains.

   The result is one or more XML message files that contain the text strings in the original language.

5. Send these files to translators, requesting translated versions.

6. When you receive the translated XML message files, import them into the same namespace from which the original was exported.

   Translated and original texts coexist in the message dictionary.

7. At runtime, the REST service chooses which text to return, based on the HTTP `Accept-Language` header.

For more information, see *String Localization and Message Dictionaries*.

B.5 Enable Web Sessions with REST

For an introduction, see Using Web Sessions with REST.

To enable your REST service to use a single web session over multiple REST calls, set the *UseSession* parameter to 1 in your REST service class:

```
Parameter UseSession As Integer = 1;
```

Note that if you choose to use a session, the system uses a CSP license until the session is ended or expires and the grace period has been satisfied. If you use the default setting for `UseSession` (which is 0), then the behavior is identical to that of SOAP requests, which hold a license for 10 seconds.

B.6 Supporting CORS

For an introduction, see Supporting CORS in REST Services. Note that the details of supporting CORS are slightly different when you create a web service manually as described on this page.

B.6.1 Modifying a REST Service to Use CORS

To specify that your REST service supports CORS, modify the REST service class as follows and then recompile it.

1. Specify a value for the *HandleCorsRequest* parameter.

   To enable CORS header processing for all calls, specify the *HandleCorsRequest* parameter as 1:

   ```
   Parameter HandleCorsRequest = 1;
   ```

   Or, to enable CORS header processing for some but not calls, specify the *HandleCorsRequest* parameter as `""` (empty string):

   ```
   Parameter HandleCorsRequest = "";
   ```

   (If *HandleCorsRequest* is 0, then CORS header processing is disabled for all calls. In this case, if the REST service receives a request with CORS headers, the service rejects the request. This is the default.)

2. If you specified *HandleCorsRequest* parameter as `""`, edit the URLMap XData block to indicate *which* calls support CORS. Specifically, for any `<Route>` that should support CORS, add the following property name and value:

   ```
   Cors="true"
   ```

   Or specify `Cors="false"` in a `<Route>` element to disable CORS processing.

If a REST service class forwards a REST request to another REST service class, the behavior of the CORS processing is determined by the class that contains the `<Route>` element that matches the given request.

B.6.2 Overriding the CORS Header Processing

Important: The default CORS header processing is not suitable for REST services handling confidential data.

The default CORS header processing does not do any filtering and simply passes the CORS header to the external server and returns the response. You may want to restrict access to origins in a domain allow list or restrict the allowed request methods. You do this by overriding the **OnHandleCorsRequest()** method in your REST service class.

For information on implementing the **OnHandleCorsRequest()** method, see Defining OnHandleCorsRequest().

Note that all URL requests that match `<Route>` elements in the **UrlMap** are processed with the single **OnHandleCorsRequest()** method that is defined in the class. If you need different implementations of the

OnHandleCorsRequest() method for different REST URL requests, you should use **Forward** to send the requests to other REST service classes.

B.7 Variation: Accessing Query Parameters

The recommended way to pass parameters to a REST service is to pass them as part of the URL path used to invoke the service (for example, `/myapi/someresource/parametervalue`). In some cases, however, it may be more convenient to pass the parameters as query parameters (for example, `/myapi/someresource?parameter=value`). In such cases, you can use the *%request* variable to retrieve the parameter values. Within a REST service, the *%request* variable is an instance of %CSP.Request that holds the entire URL query. To retrieve the value of a given query parameter, use the following syntax:

```
$GET(%request.Data(name,1),default)
```

Where *name* is the name of the query parameter and *default* is the default value to return. Or, if the same URL holds multiple copies of the same query parameter, use the following syntax:

```
$GET(%request.Data(name,index),default)
```

Where *index* is the numeric index of the copy you want to retrieve. For further details, see the class reference for %CSP.REST.

B.8 Example: Hello World!

The following code fragments represent an extremely simple sample of a REST service. There are three classes: helloWorld.disp, helloWorld.impl, helloWorld.hwObj. While helloWorld.disp and helloWorld.impl extend %CSP.REST to establish the REST service, helloWorld.hwobj extends both %Persistent and %JSON.Adaptor, which are helpful when using a POST method to create an object.

```
Class helloWorld.disp Extends %CSP.REST
{

Parameter HandleCorsRequest = 0;

XData UrlMap [ XMLNamespace = "https://www.intersystems.com/urlmap" ]
{
<Routes>
    <Route Url="/hello" Method="GET" Call="Hello" />
    <Route Url="/hello" Method="POST" Call="PostHello" />
</Routes>
}

ClassMethod Hello() As %Status
{
    Try {
        Do ##class(%REST.Impl).%SetContentType("application/json")
        If '##class(%REST.Impl).%CheckAccepts("application/json") Do
##class(%REST.Impl).%ReportRESTError(..#HTTP406NOTACCEPTABLE,$$$ERROR($$$RESTBadAccepts)) Quit
        Set response=##class(helloWorld.impl).Hello()
        Do ##class(%REST.Impl).%WriteResponse(response)
    } Catch (ex) {
        Do ##class(%REST.Impl).%SetStatusCode("400")
        return {"errormessage": "Client error"}
    }
    Quit $$$OK
}

ClassMethod PostHello() As %Status
{
    Try {
        Do ##class(%REST.Impl).%SetContentType("application/json")
        If '##class(%REST.Impl).%CheckAccepts("application/json") Do
```

```
##class(%REST.Impl).%ReportRESTError(..#HTTP406NOTACCEPTABLE,$$$ERROR($$$RESTBadAccepts)) Quit
        Set response=##class(helloWorld.impl).PostHello(%request.Data)
        Do ##class(%REST.Impl).%WriteResponse(response)
    } Catch (ex) {
        Do ##class(%REST.Impl).%SetStatusCode("400")
        return {"errormessage": "Client error"}
    }
    Quit $$$OK
}
```

Note the structure of the above dispatch class: A URLMap is defined with different end points for different methods and there are class methods that dispatch those routes to an implementation class. Along the way, the class performs some error handling. Read more about reporting errors and setting status codes in the %REST.Impl documentation. Class methods that are endpoints for POST or PUT methods will need to use the special %request parameter and pass information from said parameter on to the implementation method.

```
Class helloWorld.impl Extends %CSP.REST
{

ClassMethod Hello() As %DynamicObject
{
    Try {
       return {"Hello":"World"}.%ToJSON()
    } Catch (ex) {
        Do ##class(%REST.Impl).%SetStatusCode("500")
        return {"errormessage": "Server error"}
   }
}

ClassMethod PostHello(body As %DynamicObject) As %DynamicObject
{
    Try {
        set temp = ##class(helloWorld.hwobj).%New()
        Do temp.%JSONImport(body)
        Do temp.%Save()
        Do temp.%JSONExportToString(.ret)
        return ret
    } Catch (ex) {
        Do ##class(%REST.Impl).%SetStatusCode("500")
        return {"errormessage": "Server error"}
   }
}

}
```

The above sample implementation class has two methods, one that simply returns a “Hello World” message formatted as a JSON and one that creates an object based on some inputs and then returns the contents of said object. See the documentation for %Persistent and %JSON.Adapter for more information about the use of methods like **%New()** and **%JSONImport()**, respectively.

Although the helloWorld.hwobj class could have many properties, for the sake of simplicity in this example, it only has one:

```
Class helloWorld.hwobj Extends (%Persistent, %JSON.Adaptor)
{

Property Hello As %String;

Storage Default
{
<Data name="hwobjDefaultData">
<Value name="1">
<Value>%%CLASSNAME</Value>
</Value>
<Value name="2">
<Value>Hello</Value>
</Value>
</Data>
<DataLocation>^helloWorld.hwobjD</DataLocation>
<DefaultData>hwobjDefaultData</DefaultData>
<IdLocation>^helloWorld.hwobjD</IdLocation>
<IndexLocation>^helloWorld.hwobjI</IndexLocation>
<StreamLocation>^helloWorld.hwobjS</StreamLocation>
<Type>%Storage.Persistent</Type>
}

}
```

After configuring a Web Application in the Management Portal by following the instructions in Basics of Creating a REST Service Manually, use Postman or any other REST client to send requests to the Web Application and see its responses.