First Look: Developing REST Interfaces in InterSystems IRIS

Version 2018.1
2018-01-31

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

First Look: Developing REST Interfaces in InterSystems IRIS
InterSystems Version 2018.1 2018-01-31

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

First Look: Developing REST Interfaces in InterSystems IRIS

This *First Look* helps you develop REST interfaces. 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 Data Platform™ applications.

1 Why Provide a REST Interface

If you need to access the data in an InterSystems IRIS™ database from an external system or if you want to provide a user interface for that data, you can do it by defining a REST interface. REST — or REpresentational State Transfer — is a way to retrieve, add, update, or delete data from another system using an exposed URL. REST is based on HTTP and uses the HTTP verbs POST, GET, PUT, and DELETE to map to the common Create, Read, Update, and Delete (CRUD) functions of database applications. You can also use other HTTP verbs, such as HEAD, PATCH, and OPTIONS, with REST.

REST is one of many ways to share data between applications, so you may not always need to set up a REST service if you choose to communicate directly using another protocol, such as TCP, SOAP, or FTP. But using REST has the following advantages:

- REST typically has a small overhead. It typically uses JSON which is a light-weight 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. REST is typically much simpler to use than SOAP, which uses XML and has more overhead.
- By defining an interface in REST, it is easy to minimize the dependencies between the client and database server. This allows you to update your user interface without impacting your database implementation. You can also update the database implementation without impacting the user interface or any external applications that access the REST API.

2 How to Define REST Interfaces in InterSystems IRIS

Before defining REST interfaces, you should understand how a REST call flows through InterSystems IRIS. First consider the parts of a REST call such as:

```
GET http://localhost:57772/rest/coffeemakerapp/coffeemakers
```

This consists of the following parts:

- GET — this is the http verb.
- http: — this specifies the communication protocol.
- //localhost:57772 — this specifies the server and port number.
- /rest/coffeemakerapp/coffeemakers — this is the main part of the URL that identifies the resource that the REST call is directed to. In the following discussion, URL refers to this part of the REST call.

Note: Although this *First Look* uses the web server installed with InterSystems IRIS (using port 57772), you should replace it with a commercial web server for any code that you deploy. The web server installed with InterSystems IRIS is intended only for temporary use in developing code and does not have the robust features of a commercial web server.

When a client application makes a REST call:

1. InterSystems IRIS directs it to the web application that corresponds to the URL. For example, a URL starting with /coffeemakerapp would be sent to the application handling coffee makers and a URL starting with /api/docdb would be sent to the web application handling the Document Data Model.
2. The web application directs the call to a method based on the HTTP verb and any part of the URL after the section that identifies the web application. It does this by comparing the verb and URL against a structure called the URLMap.
3. The method uses the URL to identify the resource that the REST call is specifying and performs an action based on the verb. For example, if the verb is GET, the method returns some information about the resource; if the verb is POST, the method creates a new instance of the resource; and if the verb is DELETE, the method deletes the specified resource. For POST and PUT verbs, there is typically a data package, which provides more information.
4. If the method performs the requested action independently, it returns a response message to the client application.
5. But if the method is part of a production, creates an instance of the business service class and sends along an object with any appropriate data. The business service sends the data to a business process or business operation. After the business process or operation returns the response, the method returns the response to the client application.

Defining a REST interface requires:

- Defining the web application — you typically do this using the Management Portal. The section "Trying to Define a REST Interface for Yourself" describes how to do this.
- Creating a subclass of %CSP.REST and defining the UrlMap.
- Coding the methods that handle the REST call.
- And, optionally, creating a business service to direct the call to a production.

This *First Look* uses a sample application, coffeemakerapp, that accesses a database of coffee makers. Each record describes a coffee maker, including information such as its name, brand, and price. The coffeemakerapp provides REST interfaces to get information about the coffee makers, create a new record in the database, update an existing record, or delete a record. In a later section, this *First Look* tells you how to get the sample code from https://github.com/intersystems/FirstLook-REST.

2.1 Creating a Subclass of %CSP.REST and Defining the URLMap

Here is the first part of the demo.CoffeeMakerRESTServer class definition. It extends the %CSP.REST class.

```
Class
demo.CoffeeMakerRESTServer
Extends
%CSP.REST
 {
  Parameter
HandleCorsRequest = 1

  XData
UrlMap [ XMLNamespace
= "http://www.intersystems.com/urlmap"
]
  {
    <Routes>
      <Route
Url="/test"
Method="GET"
Call="test"/>
      <Route
```

```
Url="/coffeemakers"
Method="GET"
Call="GetAll"
/>
      <Route
Url="/coffeemaker/:id"
Method="GET"
Call="GetCoffeeMakerInfo"
/>
      <Route
Url="/newcoffeemaker"
Method="POST"
Call="NewMaker"
/>
      <Route
Url="/coffeemaker/:id"
Method="PUT"
Call="EditMaker"
/>
      <Route
Url="/coffeemaker/:id"
Method="DELETE"
Call="RemoveCoffeemaker"/>
...
   </Routes>
  }
```

Look at the Route elements. Each has three properties:

- Url — this identifies the REST URL that can be handled by this Route. Since IRIS directs URLs starting with /rest/coffeemakerapp, this property specifies the part of the URL immediately after that. If the Url property is /coffeemakers, this Route handles URLs starting with /rest/coffeemakerapp/coffeemakers.
- Method — this identifies the verb that the Route handles. Note that the last two lines have the same value for Url, /coffeemaker/:id. The Route with the PUT method will handle PUT verbs with a URL starting with /rest/coffeemakerapp/coffeemaker/:id and the Route with the DELETE method will handle DELETE verbs with the same starting URL.
- Call — specifies the method to call to process this REST call. The method is passed the complete URL and any data so it can base its response on the URL.

The part of the Url value that starts with a : represents a wildcard. That is /coffeemaker/:id will match /coffeemaker/5, /coffeemaker/200, and even /coffeemaker/XYZ. The called method will get passed the value of :id in a parameter. In this case, it identifies the ID of the coffee maker to update (with PUT) or delete.

The Url value has additional options that allow you to forward the REST URL to another instance of a %CSP.REST subclass, but you don't need to deal with that in this *First Look*. The HandleCorsRequest parameter specifies whether browsers should allow Cross-origin Resource Sharing (CORS), which is when a script running in one domain attempts to access a REST service running in another domain, but that is also an advanced topic.

2.2 Coding the Methods

The GetAll method retrieves information about all coffee makers. Here is its code:

```
ClassMethod GetAll() As %Status
{
  Set tArr = []
  Set rs = ##class(%SQL.Statement).%ExecDirect(,"SELECT * FROM demo.coffeemaker")
  While rs.%Next() {
    Do tArr.%Push({
      "img":              (rs.%Get("Img")),
      "coffeemakerID":    (rs.%Get("CoffeemakerID")),
      "name":             (rs.%Get("Name")),
      "brand":            (rs.%Get("Brand")),
      "color":            (rs.%Get("Color")),
      "numcups":          (rs.%Get("NumCups")),
      "price":            (rs.%Get("Price"))
    })
  }

  Write tArr.%ToJSON()
  Quit $$$OK
}
```

Points to note about this method:

- There are no parameters. Whenever this method is called it executes an SQL statement that selects all records from the demo.coffeemaker database.
- For each record in the database, it appends the values to an array as name, value pairs.
- It converts the array to JSON and returns the JSON to the calling application by writing the JSON out to stdout.
- Finally, it quits with a success.

The NewMaker() method has no parameters, but has a JSON payload that specifies the coffee maker to create. Here is its code:

```
ClassMethod NewMaker() As %Status
{
  If '..GetJSONFromRequest(.obj) {
    Set %response.Status = ..#HTTP400BADREQUEST
    Set error = {"errormessage": "JSON not found"}
    Write error.%ToJSON()
    Quit $$$OK
  }

  If '..ValidateJSON(obj,.error) {
    Set %response.Status = ..#HTTP400BADREQUEST
    Write error.%ToJSON()
    Quit $$$OK
  }

  Set cm = ##class(demo.coffeemaker).%New()
  Do ..CopyToCoffeemakerFromJSON(.cm,obj)

  Set sc = cm.%Save()

  Set result={}
  do result.%Set("Status",$s($$$ISERR(sc):$system.Status.GetOneErrorText(sc),1:"OK"))
  write result.%ToJSON()
  Quit sc
}
```

Points to note about this method:

- First it tests if the payload contains a valid JSON object by calling the GetJSONFromRequest() and ValidateJSON() methods defined later in the class.
- Then it uses the JSON object to create a new demo.coffeemaker and then saves the record in the database.
- It returns the status by writing it to stdout.

Finally, the RemoveCoffeemaker() method shows how the :id part of the Url is passed to the method as a parameter:

```
ClassMethod RemoveCoffeemaker(id As %String) As %Status
{
  Set result={}
  Set sc=0

  if id'="",##class(demo.coffeemaker).%ExistsId(id) {
    Set sc=##class(demo.coffeemaker).%DeleteId(id)
    do result.%Set("Status",$s($$$ISERR(sc):$system.Status.GetOneErrorText(sc),1:"OK"))
  }
  else  {
    do result.%Set("Status","")
  }

  write result.%ToJSON()

    quit sc
}
```

In summary, the methods specified by the Route Call property handles the REST call by:

- Getting any :parameter as a call argument.
- Accessing the payload through obj value.

- Returning the response to the client application by writing it to stdout.

2.3 Creating a Business Service for a Production

An InterSystems IRIS production is an interoperability framework for rapid connectivity and the development of new connectable applications. The production provides built-in connections to a wide variety of message formats and communications protocols. You can easily add other formats and protocols and use a graphic interface to define business logic and message transformations. Productions provide persistent storage of messages, which allow you to audit whether a message is successfully delivered. A production consists of business services, processes, and operations. Business services connect with external systems and receive messages from them. Business processes allow you to define business logic including routing and message transformation. Business operations connect with external systems and send the messages to them.

If you are connecting a system that sends REST messages to one that receives messages using another protocol such as FTP, TCP, or SOAP, you can use a production with a REST business service and a different protocol business operation. Typically, the business process is responsible for converting the messages, but you can also convert the message in the code you method you create to handle the REST call.

When designing your business service and production, one important decision is whether to convert the JSON to a registered object. Although both JSON and a registered object can store the same data, there are signficant differences:

- A JSON object is stored as a string. You can get the values of the data in the JSON object by using methods that parse the string.
- A registered object has defined properties. You can get the value of any property directly without examining the entire object.
- Business rules in business processes can be used to route registered objects by accessing property values.
- There is a resource cost in converting a JSON object to a registered object.

In general, if you do not need to access the values of the individual elements of the JSON object, you should not convert it to a registered object. But if you do need to convert it to a registered object, it is better to do it earlier in the production than later. Ideally this conversion should be done in the method that handles the REST call. In the Coffee Maker sample, the JSON object is not converted to a registered object. This makes it possible to have a very simple business service that is used by all of the REST methods. The business service definition is:

```
Class
demo.RESTCoffeeMakerService
Extends
Ens.BusinessService
{
  Method OnProcessInput(pInput As demo.GenericCoffeeMakerRequest, Output pOutput As
demo.GenericCoffeeMakerResponse) As %Status
  {
    set tSC = $$$OK
    try {
      set tSC = ..SendRequestSync("demo.CoffeeMakerBPL", pInput, .pOutput)
      $$$ThrowOnError(tSC)
     }
    catch ex
    {
      set tSC = ex.AsStatus()
    }
    return tSC
  }
}
```

This business service extends Ens.BusinessService and only overrides one method, OnProcessInput(). This is the method that receives the incoming message. This method sends the message to its target demo.CoffeeMakerBPL. The business service is called with a demo.GenericCoffeeMakerRequest message, which has the following definition:

```
Class demo.GenericCoffeeMakerRequest Extends Ens.Request
{
Property Method As %String;
Property URL As %String;
Property Argument1 As %String;
Property Payload As demo.tempCoffeemaker;
...
```

The properties contain the information from the REST call:

- Method — identifies the HTTP verb.
- URL — contains the URL string.
- Argument1 — has the value of the URL element defined with a : in the URL map, such as :ID. This property can represent different elements for different calls, but if any call has more than one of these : elements, then you need to define an additional properties, such as Argument2 and Argument3, until you have enough arguments to handle that call.
- Payload — contains the data payload, such as the JSON coffee maker specified in the newcoffeemaker REST call.

If there was a separate business service for each different REST call, you would not have to include the method and URL because the URLMap selected the method to call based on these elements. In some cases, you could design your REST service in this way. In these cases, you may only need to pass the arguments and the payload to the business service. But the coffeemaker sample uses a simpler design, where all of the different REST calls are sent to a single business service. Consequently, this business service needs to access the method and URL to determine the action to take.

Methods in the %CSP.REST subclass that call business services need to do the following:

1. Create a new instance of the business service.
2. Populate the request that will be sent to the business service.
3. Call the ProcessInput() method from the service object. This sends the message to the business service.
4. Set the response message's header and metadata information.
5. And, finally, write the response back to the client.

The methods in the coffee maker sample use a helper method , CallInterface, to perform these functions. Although this is an excellent programming practice, it makes it a little harder to see what's happening. Here is a method that doesn't use a helper method to get information about all coffee makers using a production to get the information:

```
ClassMethod GetCoffeeMakerInfoFromInterface(id As %String) As %Status
{
  set tSC = $$$OK
  try {
    // Instantiate business service
    set tSC = ##class(Ens.Director).CreateBusinessService("demo.RESTCoffeeMakerService",.tService)
    $$$ThrowOnError(tSC)
    //Pass aloong input from url to ProcessInput.
    set request = ##class(demo.GenericCoffeeMakerRequest).%New()
    set request.URL = "/coffeemaker/:id/int"
    set request.Method = "GET"
    set request.Argument1 = id
    set request.Payload = ""
    set tSC = tService.ProcessInput(request, .output)

    $$$ThrowOnError(tSC)

    do %response.SetHeader("ContentType", "application/json")
    #Dim output As demo.GendericCoffeeMakerResponse

    //Write JSON response back to client application
    write output.JSONResponse
  }
  catch ex {
    set tSC = ex.AsStatus()
  }
  return tSC
}
```

This completes a quick tour of the coffee maker sample code. Although this tour has ignored some of the details of the code, it has highlighted the important elements so that you can understand the overall logic.

3 Trying to Define a REST Interface for Yourself

This section shows you step-by-step how to use the coffee maker application to handle REST calls. It starts with getting your system ready, including downloading the sample code from github, and takes you through the steps required to build the code and create the environment in the Management Portal.

3.1 Getting Your System Ready

Before creating this example, you should do the following:

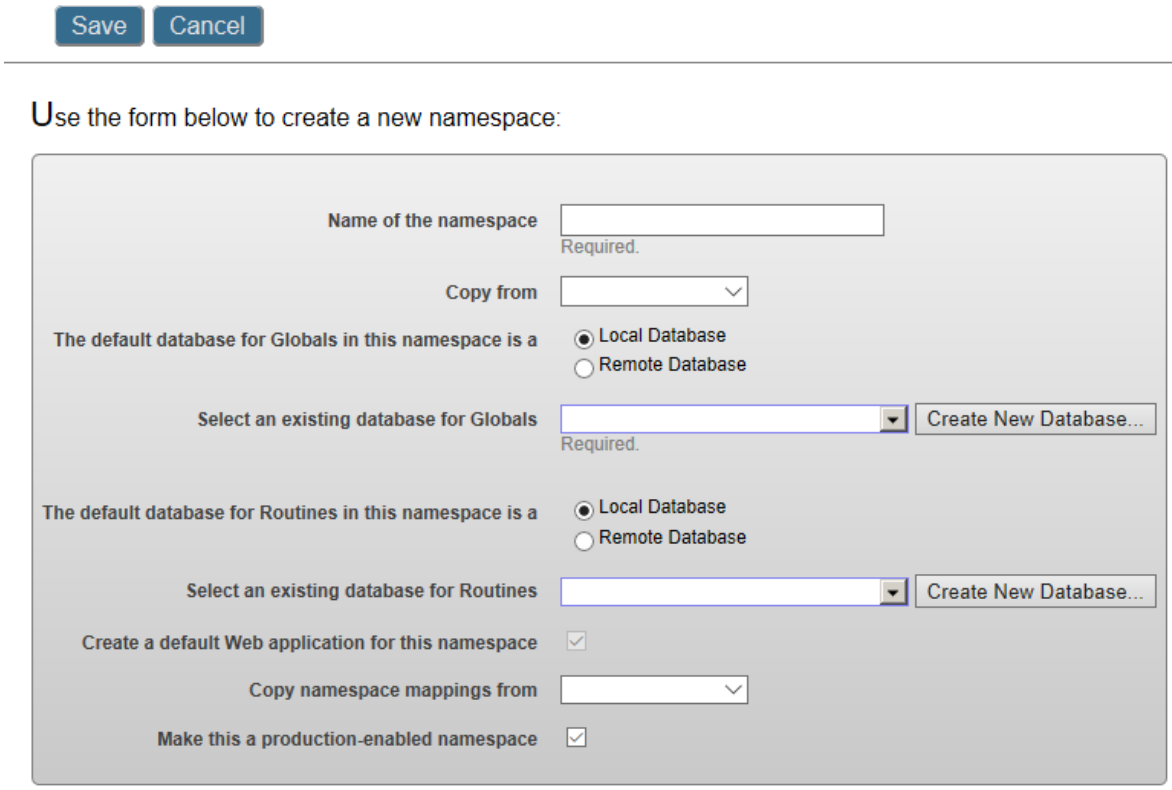
- Install a running, licensed instance of InterSystems IRIS.
- Install a REST API application such as Postman, Chrome Advanced REST Client, or cURL. You will use it to generate test REST HTTP commands to show that your REST interfaces are working..
- Clone or download the FirstLook-REST sample code from github: https://github.com/intersystems/FirstLook-REST. This sample demonstrates REST APIs for a coffee maker database. If you're not familiar with github:
  1. Go to https://github.com/intersystems/FirstLook-REST in a web browser.
  2. Select **Clone or download**.
  3. Select Download README-REST-master.zip.
  4. Extract the files from the zip archive.
  5. Using a text editor, open the README.md file and follow the instructions in it. Details of creating a namespace and web applications are in the following section.

3.2 Creating a Production-Enabled Namespace

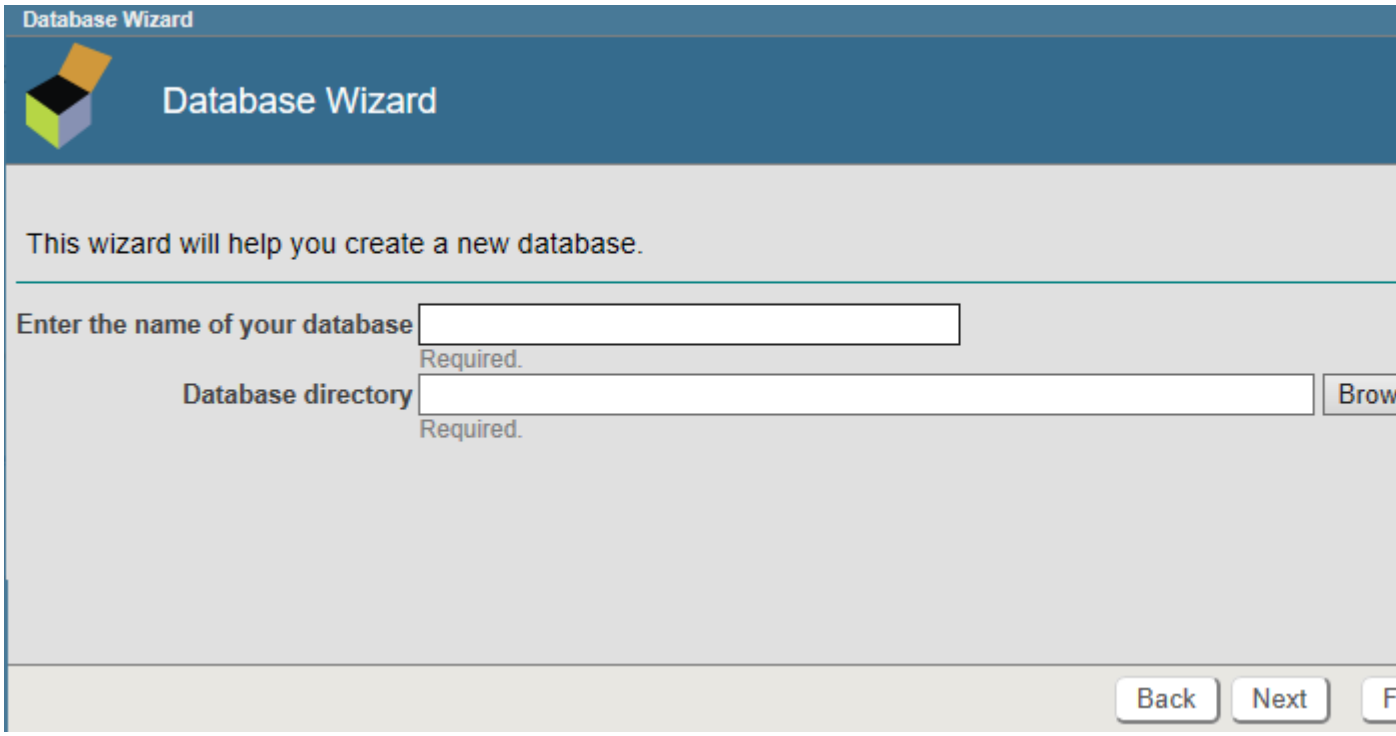
In order to create a production, you must have a production-enabled namespaces. The namespaces created when you first install InterSystems IRIS are not production-enabled. You will create a production-enabled namespace in this section. If you already have a production-enabled namespace, you can skip this step. Although the first REST interfaces you will work with in this section do not use productions, the last steps do and the coffee maker sample REST APIs include production business services.

In the Management Portal:

1. Select **System Administration** > **Configuration** > **System Configuration** > **Namespaces** to go to the **Namespaces** page.
2. On the **Namespaces** page, select **Create New Namespace**. This displays the **New Namespace** page:



3. On the **New Namespace** page, enter the name for the new namespace, such as `SAMPLES`.
4. Next to the **Select an existing database for Globals** drop-down menu, select **Create New Database.** This displays the **Database Wizard:**



5. On the first page of the **Database Wizard**, in the **Enter the name of your database** field, enter the name of the database you are creating, such as `testdb`. Enter a directory for the database, such as `C:\InterSystems\IRIS\mgr\samplesdb`. On that page, select **Next**.
6. On the next page, select **Finish**.
7. Back on the **New Namespace** page, in the **Select an existing database for Routines** drop-down menu, select the database you just created.
8. Ensure that the `Make this a production-enabled namespace` check box is selected.
9. Select **Save** near the top of the page and then select **Close** at the end of the resulting log.

You have now created a production-enabled namespace.

For more details about creating a namespace and its associated database, see "Create/Modify a Namespace" in the "Configuring InterSystems IRIS" chapter of the *InterSystems IRIS System Administration Guide*. For background information, see "Namespaces and Databases" in the *Orientation Guide for Server-Side Programming*.

3.3 Build the Sample Code

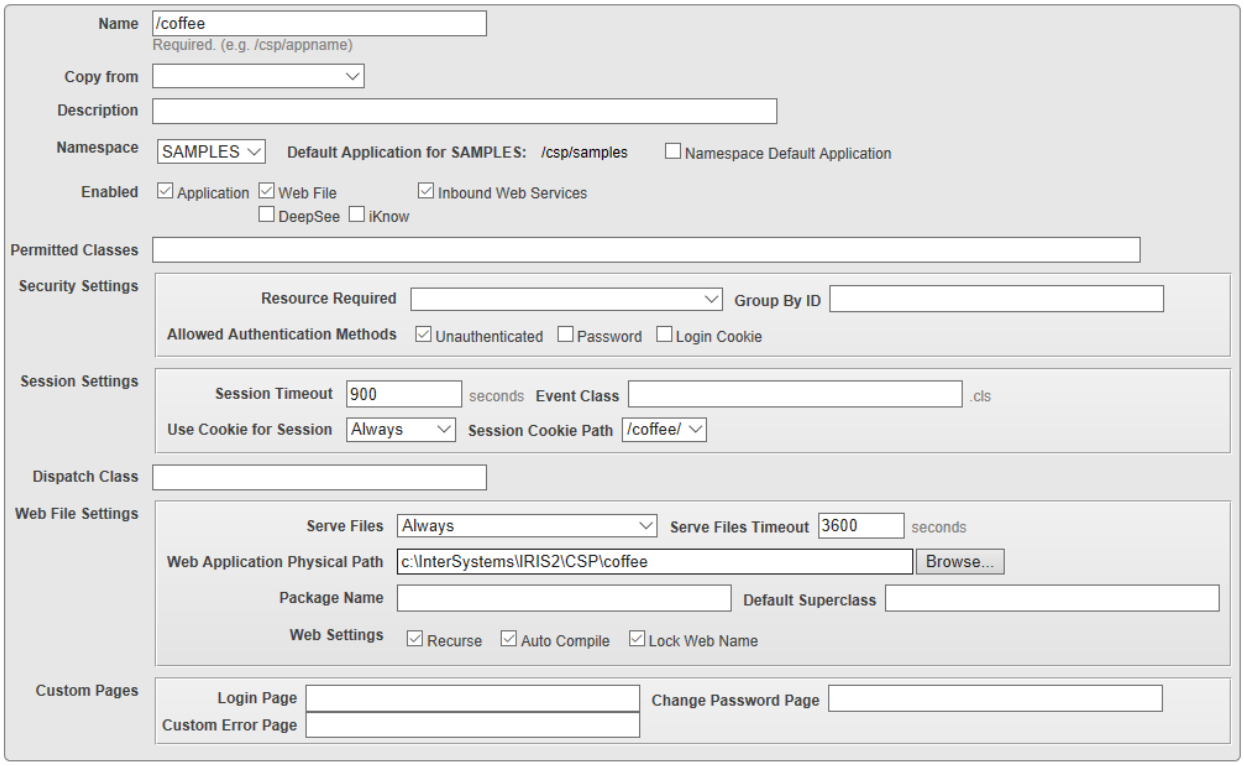
Build the sample code. Details are in the README.md provided in the github repository.

3.4 Defining Web Applications

In the Management Portal, first you create a web application to contain the application. This is not the web application that will process the REST requests.

1. Select **System Administration** > **Security** > **Applications** > **Web Applications**.
2. Select **Create New Web Application** and enter the following values:
   a. **Name**: `/coffee`
   b. **Namespace**: the name of the production-enabled namespace
   c. **Web Application Physical Path** : *install-dir*`\CSP\coffee\`

   Your New Web Application form should be similar to:

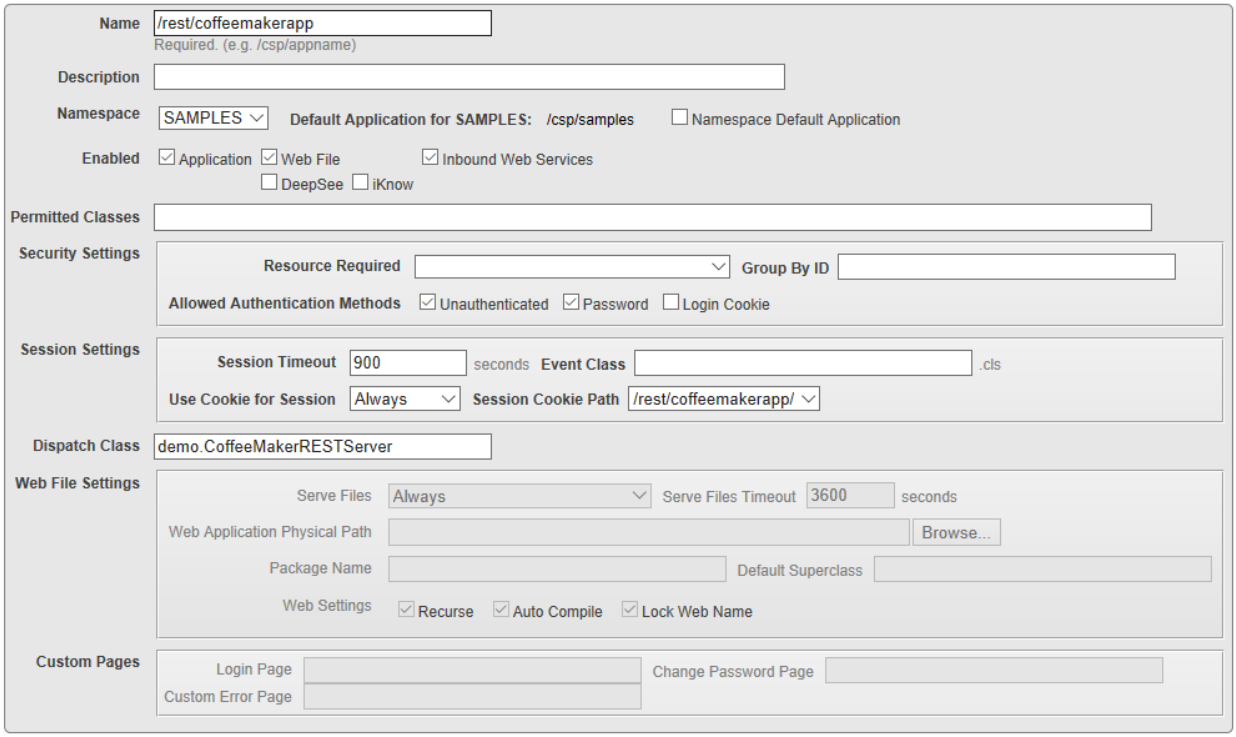


3. Select **Save**.

Now you create a second web application. This is the one that handles the REST calls.

1. Select **System Administration** > **Security** > **Applications** > **Web Applications**.
2. Select **Create New Web Application** and enter the following values:
   a. **Name**: `/rest/coffeemakerapp` — this specifies the URLs that will be handled by this web application. InterSystems IRIS will direct all URLs that begin with `/rest/coffeemakerapp` to this web application.
   b. **Namespace**: the name of the production-enabled namespace
   c. **Allowed Authentication Methods**: select both **Unauthenticated** and **Password** check boxes.
   d. **Dispatch Class**: `demo.CoffeeMakerRESTServer` — this is the class that defines the URLMap.

   Your New Web Application form should be similar to:



3. Select **Save**.
4. Ensure you are in the production-enabled namespace you are using for this sample and select **Interoperability** > **List** > **Productions** and open demo.CoffeeMakerProduction.
5. Select **Start** to start the production.

3.5 Accessing the REST Interfaces

The CoffeeMaker REST application is now working. You will enter REST commands to access the coffee maker database. In your REST API tool, such as Postman, follow these steps:

1. Specify the following REST call to get a list of coffee makers in the database:
   - HTTP Action: GET
   - URL: http://*server*:*port*/rest/coffeemakerapp/coffeemakers
   - Username and password for your InterSystems IRIS account

   The call returns a list of coffeemakers, such as:


```
[{"img":"img/Coffee1 (2).png","coffeemakerID":"1","name":"Regular coffee pot2","brand":"Coffee Road","color":"Red","numcups":1,"price":21.98},
{"img":"img/coffee2 (2).png","coffeemakerID":"2","name":"Single Cup Take-away","brand":"Momma's Kitchen","color":"Black","numcups":1,"price":23.98},
...
{"img":"img/coffee13.png","coffeemakerID":"39","name":"Double Espresso","brand":"Coffee Road","color":"Blue","numcups":2,"price":71.73}]
```


2. Specify the following REST call to delete the coffee maker with ID=2:
   - HTTP Action: DELETE
   - URL: http://*server*:*port*/rest/coffeemakerapp/coffeemaker/2
   - Username and password for your InterSystems IRIS account

The call returns a success status:

```
{"Status":"OK"}
```

3. Specify the following REST call to add a new coffee maker:
   - HTTP Action: POST
   - URL: http://*server*:*port*/rest/coffeemakerapp/newcoffeemaker
   - Username and password for your InterSystems IRIS account
   - Input data:

     ```
     {"img":"img/coffee85.png","coffeemakerID":"99","name":"Double Dip","brand":"Coffee+","color":"Blue","numcups":2,"price":71.73}
     ```

   Although the data contains a value for coffeemakerID, that is a calculated field and a new value is assigned when the record is addded. The call returns a success status:

   ```
   {"Status":"OK","Message":"New maker saved with ID 39"}
   ```

3.6 Using REST with Productions

The CoffeeMakerApp REST calls to a production are very similar to the database access calls. The only difference in the URL is that `/int` is appended to the URL. For example, to get the coffee maker with ID equal to 3 by enter the following REST call:

- HTTP Action: GET
- URL: http://*server*:*port*/rest/coffeemakerapp/coffeemaker/3/int
- Username and password for your InterSystems IRIS account

The call returns the specified coffee maker:

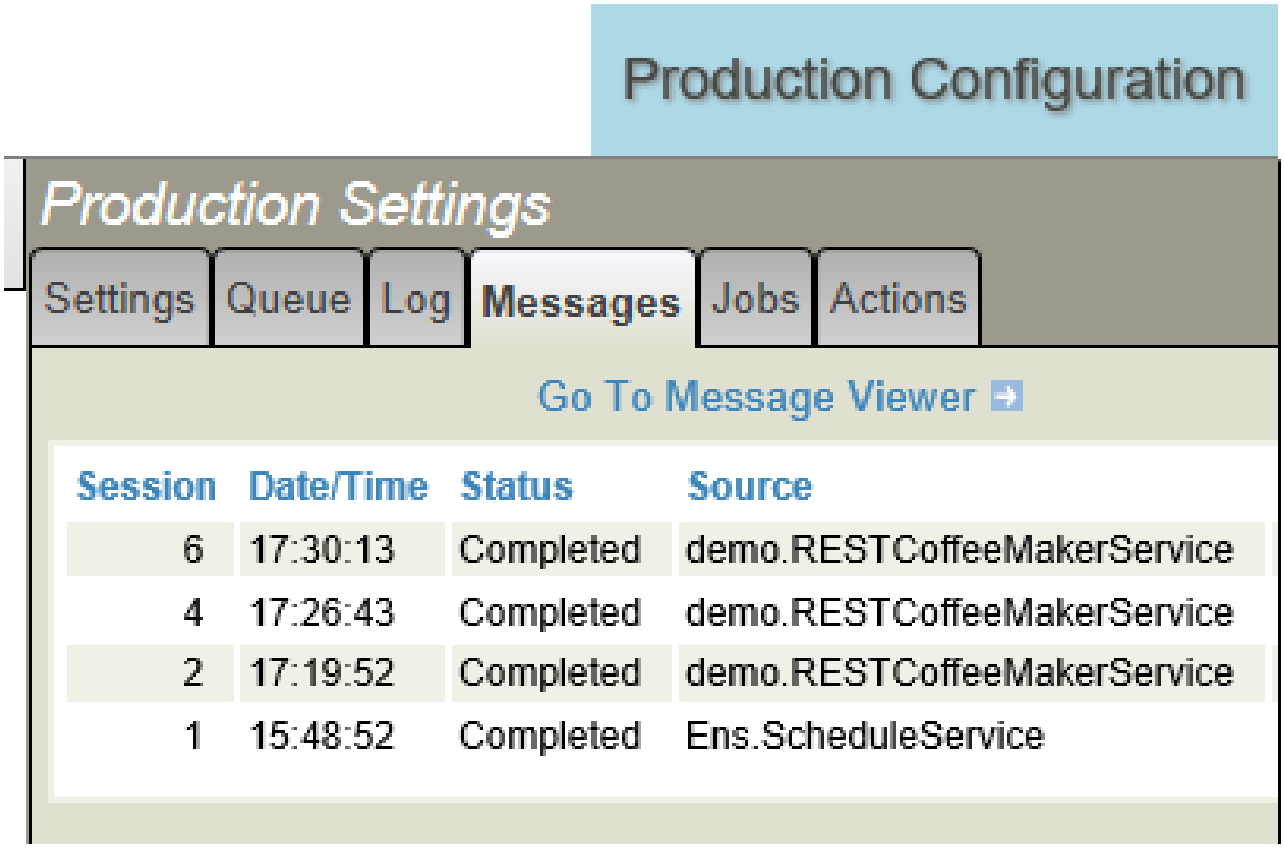
```
{"CoffeemakerID":"3","Name":"Double Espresso","Brand":"Coffee Road","Color":"Blue","NumCups":2,"Price":41.73,"Img":"img/coffee3.png"}
```

You can add a coffee maker by entering the following REST call:

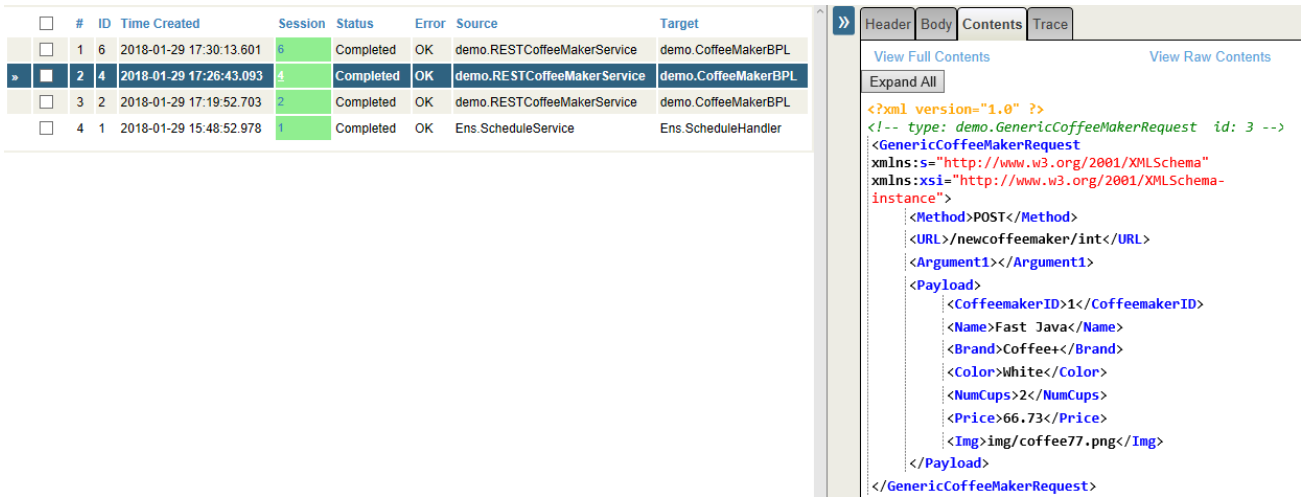
- HTTP Action: POST
- URL: http://*server*:*port*/rest/coffeemakerapp/newcoffeemaker/int
- Username and password for your InterSystems IRIS account
- Input data:

  ```
  {"img":"img/coffee77.png","coffeemakerID":"99","name":"Fast Java","brand":"Coffee+","color":"White","numcups":2,"price":66.73}
  ```

To see the messages in the production, select the **Messages** tab on the Production Configuration page. The messages are displayed as shown by:



To see the contents of a message, select **Go To Message Viewer**. Select **Search** in the message viewer, select a message, and select the **Contents** tab. The Message Viewer shows you the following:



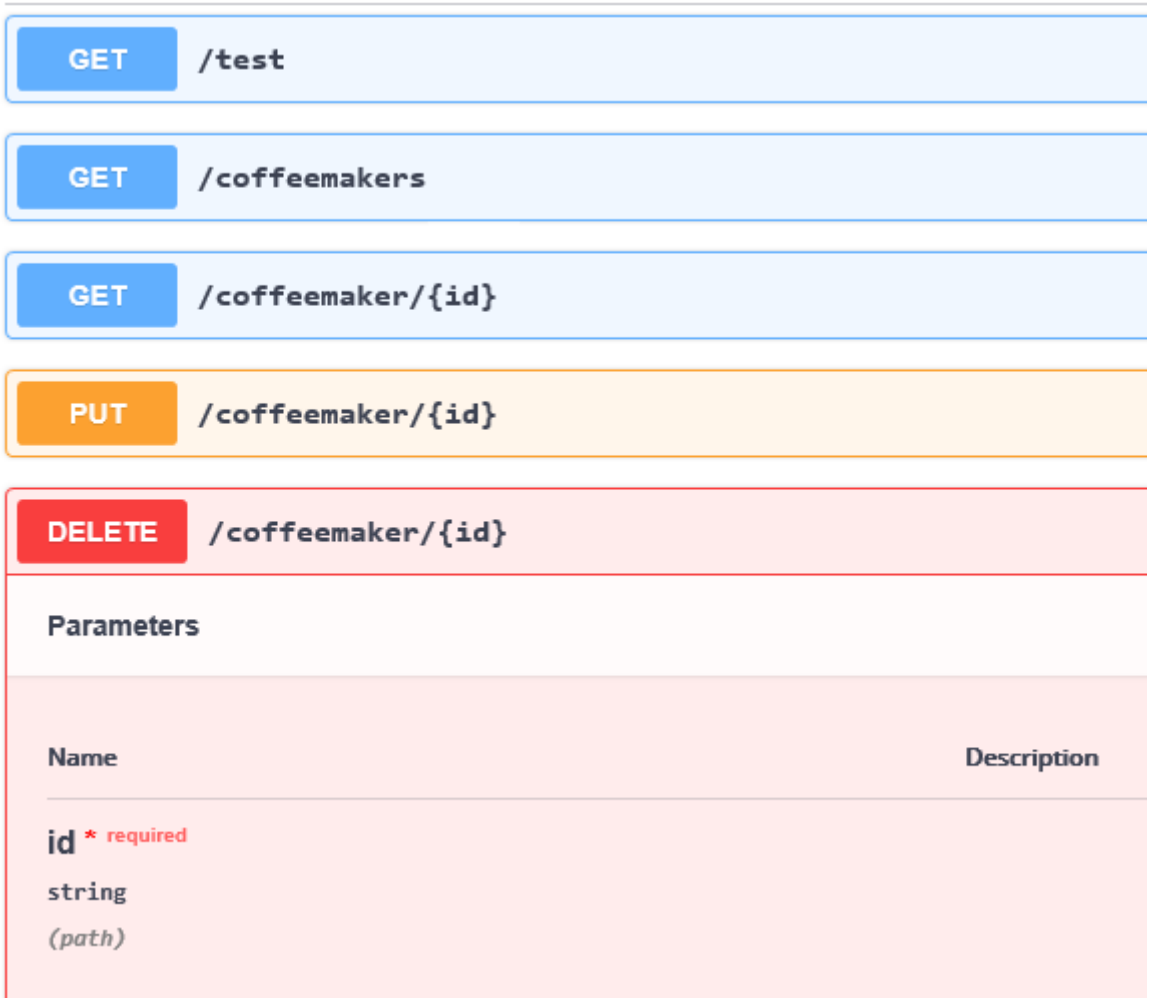
3.7 Documenting REST Interfaces

When you provide REST interfaces to developers you should provide documentation so that they know how to call the interfaces. You can use the Open API Spec to document REST interfaces and a tool, such as Swagger to edit and format the documentation. InterSystems is developing a feature to support this documentation. This release contains a feature in API Management that generates the document framework for your REST APIs. You still need to edit the generated documentation to add comments and additional information, such as content of arguments and HTTP return values.

To generate the documentation for the CoffeeMakerApp REST sample, enter the following REST call:

- HTTP Action: GET
- URL: http://*server*:*port*/api/mgmnt/v1/*namespace*/spec/rest/coffeemakerapp/
- Username and password for your InterSystems IRIS account

You can paste the output from this call into the swagger editior. It converts the JSON to YAML (Yet Another Markup Language) and displays the doc. You can use the swagger editor to add more information to the documentation. The swagger editor displays the documentation as shown in the following:



4 For More Information on InterSystems IRIS and REST

See the following for more information on creating REST services in InterSystems IRIS:

- Setting Up RESTful Services is an InterSystems online class that uses the same coffee maker application as this *First Look*, but goes into more detail. You need to be logged into learning.intersystems.com to take this course. If you don't have an account, you can create one.
- Creating REST Services and Using REST in Productions