Creating REST Services in Caché

Version 2018.1
2020-11-13

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

Creating REST Services in Caché
Caché Version 2018.1 2020-11-13

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

About This Book

This book describes, to programmers, how to create Caché REST services. It includes the following sections:

- REST Service Overview
- Creating REST Services
- Supporting CORS in REST Services
- DocServer Sample

For a detailed outline, see the table of contents.

For more information, try the following sources:

- *Using Caché Server Pages (CSP)* describes how to create web applications that consist of CSP pages.
- *Using Caché Internet Utilities* includes information on using HTTP responses.

For general information, see the *InterSystems Documentation Guide*.

1

REST Service Overview

Caché provides the capabilities to implement a REST service that can be invoked from a REST call.

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

- REST is an architectural style rather than a clearly defined 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, and DELETE.
- REST typically has a small overhead. Although it can use XML to describe data, 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.

To implement a REST service, you must extend the abstract class %CSP.REST.

2

Creating REST Services

This chapter describes how to build Caché REST services using the %CSP.REST class. This class enables you to create a REST service.

This chapter contains the following sections:

2.1 The %CSP.REST Class

The %CSP.REST class, which is a subclass of %CSP.Page allows you to implement REST services. It provides the ability to:

- Define URL maps that specify the Caché method that is executed for a REST URL and HTTP method.
- Specify whether each REST call is executed under its own CSP session or shares a single session with other REST calls.
- Optionally, override error handling methods.

To implement a REST service, you extend the %CSP.REST class. Your extension of the class provides the URLMap, can optionally set the **UseSession** parameter, and provides class methods to perform the REST operations. You can define more than one subclass of %CSP.REST in a namespace. Each subclass that has its own entry point must have its own CSP web application. You define the CSP web application and specify its security in the **Web Application** page (click **System Administration** > **Security** > **Applications** > **Web Applications**). When you define the CSP web application, you set the **Dispatch Class** to the name of the custom subclass of %CSP.REST and specify the first part of the URL for the REST call as the name of the application.

2.2 Creating the URL Map for REST

The XDATA UrlMap 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 subclass of %CSP.REST 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 subclasses of %CSP.REST, each of which handles a set of related services. Then the subclass of %CSP.REST that handles the web application simply forwards the REST call to the appropriate subclass.

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 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 statement forwards all calls with the specified prefix to its associated %CSP.REST subclass, 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.

Caché compares the incoming REST URL with the Route URL property or the Map Prefix property. It starts at the first Route or Map element and continues to test each following element until it finds a match. Once it finds a match it either sends the incoming call to the call specified in the Route or forwards the URL to the class specified in the Map. In both cases, it ignores any elements in the Routes after the matching element; consequently, the order of the elements in the Routes is significant. If an incoming URL could match multiple elements of the Routes, Caché uses the first matching element and ignores any subsequent possible matches.

2.2.1 URLMap with Route Elements

Caché compares the incoming URL and the HTTP request method to each Route element in Routes. 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 the class that subclasses %CSP.REST, but you can explicitly specify a class method in any Caché class.

For example, the DocServer sample defines the following Route:

```
XData UrlMap
{
<Routes>
  <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, has a value of `"true"` or `"false"` and controls CORS header processing. See “Configuring a REST Service to Use CORS” for details on using CORS.

The complete REST URL consists of the following pieces:

- Server name and port of the Caché server. In this chapter, the server name and port `http://localhost:57772/` is used in the examples.
- Name of the web application as defined on the **Web Application** page (click **System Administration** > **Security** > **Applications** > **Web Applications**). For the DocServer sample, the name is `/csp/samples/docserver`, but you can specify any text that is allowed in a URL.
- The remainder of the REST URL is defined by the Route Url element. If a segment of the Url element 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:57772
```

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, the REST call:

```
http://localhost:57772/csp/samples/docserver/class/samples/Cinema.Review
```

calls the GetClass method and specifies “samples” and “Cinema.Review” as the parameter values. The definition of the GetClass method starts with:

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

You can define a different method for each HTTP request method for a single URL. For example, you could define the following:

```
<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 the DocServer sample, both HTTP request methods are handled by a single Call method. The Route definitions in the sample is

```
<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. If this method needs to distinguish between a GET and a POST operation, it can do this by examining the CSP request object, which contains the text of the URL.

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.

2.2.1.1 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” in *Using ObjectScript*.)

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"/>
  ```

2.2.2 URLMap with Map Elements

Caché compares the incoming URL to the prefix in each Map element in Routes. It forwards the incoming REST call to the %CSP.REST subclass 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 parts:

- Prefix—specifies the segment of the URL to match. The incoming URL typically has other segments after the matching segment.
- Forward—specifies another subclass of %CSP.REST 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 subclasses of %CSP.REST: 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:

- Server name and port of the Caché server, such as `http://localhost:57772/`.
- 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`.
- One of the prefixes define in the Map elements.
- The remainder of the REST URL. This is the URL that will be processed by the %CSP.REST subclass that receives the forwarded REST request.

For example, the following REST call:

```
http://localhost:57772/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:57772/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.

2.3 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"
```

2.4 Localizing a REST Service

Any string value returned by a Caché 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.

   It may be helpful to also send the CacheMessages.dtd file, which describes the schema required by the message files. The file format, however, is quite simple and the DTD may not be necessary.

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 the article *String Localization and Message Dictionaries*.

2.5 Using a CSP Session with REST

The **UseSession** parameter controls whether Caché uses a new CSP session for each REST service call or preserves a CSP session across multiple REST service calls. 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 CSP session preserved across REST calls breaks the stateless paradigm, but there are two reasons why you might want to preserve a CSP session:

- Minimize Caché license usage—if each REST call creates a new CSP session it requires a separate Caché license. Caché frees the license after a brief timeout period. If you are making rare and infrequent REST calls, this is usually not a problem, but if you are making many REST calls in a brief time period, you can exceed the number of licenses available on your system. If you use the same CSP session for multiple REST calls, the session only consumes one license.
- 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 CSP session over multiple REST calls, set the **UseSession** parameter to 1 in your subclass of %CSP.REST that is defined as the dispatch class for a CSP web application. For example:

```
Class REST.MyServices Extends %CSP.REST {
Parameter UseSession As Integer = 1
```

2.6 Authentication in REST

If your REST service is accessing confidential data, you should use authentication. You can use either of the two following forms of authentication:

- HTTP authentication headers—this is the recommended form of authentication for REST services.
- CSP session authentication—where the username and password are specified in the URL following a question mark.

2.7 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.

3

Supporting CORS in REST Services

Cross-origin Resource Sharing (CORS) allows a script running in another domain to access a Caché REST 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 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 a Caché 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 Caché 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.

Caché supports CORS by passing the HTTP headers and allows you to configure whether a REST service allows the CORS header. You must write code that defines when to allow a CORS request. For example, you can provide a white-list containing domains that contain only trusted scripts. Caché does provides a simple default implementation for documentation purposes but it allows any CORS request. You should not enable CORS processing for confidential data using this default implementation.

To write the code that controls CORS requests, you override the **OnHandleCorsRequest()** method in your %CSP.REST subclass.

This chapter contains the following sections:

3.1 Configuring a REST Service to Use CORS

You control whether a REST service supports CORS with the %CSP.REST **HandleCorsRequest** parameter and with the **Cors** attribute on the **route** element in the **UrlMap**.

To enable or disable CORS header processing for all REST services defined in the %CSP.REST subclass, set the **HandleCorsRequest** parameter to `"true"` to enable CORS processing or to `"false"`, which is the default, to disable CORS processing.

To enable or disable CORS header processing independently for each **Route** in the **UrlMap**, set the **HandleCorsRequest** parameter to `""` (empty string) and specify `Cors="true"` in a **Route** element to enable CORS processing and `Cors="false"` in a **Route** element to disable CORS processing.

If a %CSP.REST subclass forwards a REST request to a second %CSP.REST subclass that contains a **Route** matching that REST URL, the **HandleCorsRequest** parameter in the %CSP.REST subclass containing the matching **Route** controls the behavior of the CORS processing.

If CORS processing is disabled for an incoming REST URL with CORS headers, %CSP.REST rejects the incoming request.

Important: A Caché REST service supports the OPTIONS request (the *CORS preflight request*), which is used to determine whether a REST service supports CORS. This request is executed by the CSPSystem user. This user should have READ permission on any databases used by the REST service; if not, the service will respond with an HTTP 404 error.

3.2 Overriding the OnHandleCorsRequest Method

The default implementation of the **OnHandleCorsRequest()** method 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 that are listed in a domain white list or to restrict what request methods are allowed. You do this by overriding the **OnHandleCorsRequest()** method in your %CSP.REST subclass.

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 to have different implementations of the **OnHandleCorsRequest()** method for different REST URL requests, you should use **Forward** to send the requests to different subclasses of %CSP.REST.

To implement the **OnHandleCorsRequest()** method, you must be familiar with the details of the CORS protocol. This section identifies the parts of the default **OnHandleCorsRequest()** method implementation and identifies the lines that handle the origin, credentials, header, and request method.

The following code from the **%CSP.REST.HandleDefaultCorsRequest()** method gets the origin and use it to set the response header. One possible way to handle this is to test the origin against a white list and only use it to set the response header if the domain is allowed. If it is not allowed, you can 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. Add code to test if these headers and request method are allowed. If they are allowed, use them to set the response headers.

```
#; 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)
```

Note: The **%CSP.REST.HandleDefaultCorsRequest()** method provides a simple default implementation for documentation purposes only. You should not enable CORS processing for confidential data using this default implementation.

4

DocServer Sample

The Samples namespace contains the DocServer REST sample, which returns information about the documentation for Caché classes. You can view the REST.DocServer source in Studio. The comments at the beginning of the sample describe the URLs that you use to invoke the REST services. You can invoke the sample REST services that use an HTTP GET operation using a web browser. For example, if you enter the following:

`http://localhost:`*`nnnnn`*`/csp/samples/docserver/namespaces`

The sample returns the namespaces defined in Caché. But, to invoke REST services that use other HTTP operations, such as POST, you need to use an HTTP utility, such as cURL.

Note: The docserver sample is a simple example without any security. By default, its web application is disabled to prevent unauthorized users from accessing namespace data. Consequently, before running the sample, you must enable its web application. To do this:

1. On the **Web Applications** page, select the /csp/samples/docserver application.
2. On the **General** tab, enable the web application by selecting the **Enabled Application** check box.
3. Select **Save**.