Cúram 8.2

Cúram REST API Guide

Note

Before using this information and the product it supports, read the information in Notices on page 119

Edition

This edition applies to Cúram 8.2.

Contents

Chapter 1 Integrating with external applications through REST APIs

You can connect to existing inbound Cúram REST APIs or create your own custom inbound REST APIs. You can also make outbound API requests to integrate with external applications that expose REST APIs.

Related tasks

Developing inbound REST APIs on page 47
If the existing Cúram REST APIs do not meet all of your requirements, you can create custom inbound REST APIs. For example you can create APIs to integrate mobile applications that connect Cúram with mobile users. For more information about the existing REST API resources, see the related information links.

Creating a Cúram REST API on page 47
Complete the following steps to create a simple Cúram REST API. After you create a custom REST API, you can use it to integrate with other applications, for example, a mobile app.

1.1 Integrating with inbound REST APIs

Cúram provides a collection of REST API resources and supports the creation of inbound REST APIs. You can integrate external systems, such as a mobile application, with Cúram by using the REST APIs. If the existing REST API resources do not meet your needs, you can create your own custom REST API resources.

Integration of mobile applications, web applications or external applications with Cúram

Mobile applications, web applications or any external application can integrate with Cúram using REST APIs. REST APIs support retrieving and modifying data in Cúram using the JSON notation, and are suitable for both hybrid and native mobile application development.

Review the existing REST APIs supported by Cúram to determine if they meet the needs of your mobile application. Alternatively, you can create your own custom REST APIs.

Using existing Cúram REST APIs

You can review existing Cúram REST APIs to determine if they are suitable for your requirements.

Before you begin

You must have access to a Cúram development environment.

Procedure

1. Navigate to the Swagger specification URI at *https://<host>:<port_number>/Rest/api/definitions*. A basic HTML page opens, containing links to the Swagger specification for each API version.
2. Click a link to open the Swagger specification that you require in JSON format.
3. Optional: If you have installed Swagger-enabled tools, for example Swagger UI, save the JSON document for the specific API versions that are listed at the link *https://<host>:<port_number>/Rest/api/definitions* and open this document by using the Swagger-enabled tools.
4. Review the REST APIs documented in Swagger to assess if they are suitable for your requirements.

What to do next

If you are using a tool like Swagger UI, you can try out the REST APIs to see whether they meet your requirements.

Making a basic GET request

In this example, you use a web browser to invoke an API resource GET method to retrieve a list of Person objects. Cúram REST APIs support GET, PUT, POST, and DELETE methods. The GET method is the simplest and the easiest method to request.

Procedure

1. Enter the following URL to authenticate to the Cúram server with the j_username and j_password parameters:

```
https://<host>:<port>/Rest/j_security_check?
j_username=<username>&j_password=<password
```

2. To make a GET request, enter the relevant URL in a browser, for example:

```
https://<host>:<port>/Rest/v1/persons?full_name=smith
```

This is a collection resource, which returns a list of persons represented as JSON objects.

Option	Description
https://<host>:<port>/Rest	The context path of the URL where you can access the REST APIs.
v1	The version of the REST API that you want to invoke.
persons	The resource that you want to access.
?full_name=smith	The query parameter that you are passing to the URL. This query parameter filters the list of people that are returned, and in this case is mandatory.

Results

If the request is successful, the result of the resource GET request in the browser is a JSON representation of the list of Persons found, based on the full_name query parameter value. The browser displays the result as text on the screen. For example, the following is a subset of what Cúram returns for this resource:


```
{
  "data" : [ {
    "photo" : "/vyfcommnhyyccjdy/106",
    "dateOfBirth" : "1938-04-11",
    "full_name" : "Robert Smith",
    "concern_role_id" : "106",
    "primaryAddress" : {
      "displayText" : "314, Old Road\nZinfadel\nMidway, Utah, 12346\nUnited States",
      "addressType" : {
        "tableName" : "AddressType",
        "value" : "AT1",
        "description" : "Private",
        "parentCodeTable" : null
      }
    }
  },
  {
    "photo" : "/vyfcommnhyyccjdy/101",
    "dateOfBirth" : "1964-09-26",
    "full_name" : "James Smith",
    "concern_role_id" : "101",
    "primaryAddress" : {
      "displayText" : "1074, Park Terrace\nFairfield\nMidway, Utah, 12345\nUnited
 States",
      "addressType" : {
        "tableName" : "AddressType",
        "value" : "AT1",
        "description" : "Private",
        "parentCodeTable" : null
      }
    }
  }]
}
```


This result shows an array of persons, with two entries or objects; one for James Smith and one for Robert Smith. The array of persons can be accessed using the data property.

If the request is unsuccessful or if you are not logged in to the Cúram server, you see an HTTP error response code.

Making a GET request with a path parameter

In this example, you use a web browser to invoke an API resource GET method for a specific Cúram Person object.

About this task

This example shows the usage of a path parameter in the resource URL, which supports member resources, that is, a single object representing the requested resource.

Procedure

1. Enter the following URL to authenticate to the Cúram server with the j_username and j_password parameters:

```
https://<host>:<port>/Rest/j_security_check?
j_username=<username>&j_password=<password
```

2. To make a GET request for a specific person, access the relevant URL, for example:

```
https://<host>:<port>/Rest/v1/persons/101/
```

This is a member resource, which returns a single person represented as a JSON object.

Option	Description
https://<host>:<port>/Rest	The context path of the URL where you can access the REST APIs.
v1	The version of the REST API that you want to invoke.
persons	The resource you want to access.
101	The path parameter, indicating the unique identifier for the specific person to be returned.

Results

If the request is successful, the result of the resource GET request is a JSON representation of the requested person displayed as text in the browser. For example, here is a subset of what Cúram returns for this resource:

```
{
  "photo" : "/vyfcommnhyyccjdy/101",
  "dateOfBirth" : "1964-09-26",
  "full_name" : "James Smith",
  "concern_role_id" : "101",
  "primaryAddress" : {
    "displayText" : "1074, Park Terrace\nFairfield\nMidway, Utah, 12345\nUnited
 States",
    "addressType" : {
      "tableName" : "AddressType",
      "value" : "AT1",
      "description" : "Private",
      "parentCodeTable" : null
    }
  }
}
```

If the request is unsuccessful or if you are not logged into the Cúram server, you get a HTTP error response code.

Making a GET request using the field selection query parameter

In this example, you use a web browser to invoke an API resource GET method, requesting specific properties to be returned.

About this task

This example demonstrates usage of the field selection query parameter for a GET method. Where the _fields query parameter is specified as part of a GET request, only the specified properties will be included as part of the returned JSON resource representation.

Procedure

1. Enter the following URL to authenticate to the Cúram server with the j_username and j_password parameters:

```
https://<host>:<port>/Rest/j_security_check?
j_username=<username>&j_password=<password
```

2. Make a GET request, specifying a subset of the resource properties to be returned, by adding the _fields query parameter listing the properties, to the relevant URL. For example:

```
https://<host>:<port>/Rest/v1/persons/101?
_fields=concern_role_id,fullName,photo,primaryPhoneNumber
```

   This will return only the requested person properties concern_role_id, fullName, photo and primaryPhoneNumber in the JSON resource representation.

Option	Description
https://<host>:<port>/Rest	The context path of the URL where you can access the REST APIs.
v1	The version of the REST API that you want to invoke.
persons	The resource you want to access.
101	The path parameter, indicating the unique identifier for the specific person to be returned.
_fields=concern_role_id,photo,primaryPhoneNumber	The field selection parameter, instructing API to return only these requested properties and not all the properties of person resource.

Results

If the request is successful, the requested properties are displayed as JSON text in the browser. For example, here is what Cúram returns for this resource:

```
{
  "photo" : "/abcdadksjfqhg2uq/1010000",
  "concern_role_id" : "101",
  "primaryPhoneNumber" : {
    "countryCode" : "1",
    "areaCode" : "555",
    "number" : "3477455",
    "extension" : "",
   }
}
```

The field selection query parameter will ignore any requested properties that are not part of the resource.

If the request is unsuccessful or if you are not logged into the Cúram server, you get a HTTP error response code.

Making a POST request

In this example, you create a new Note object by calling the POST method on the Notes resource.

Before you begin

You cannot make a POST request by using a web browser, as web browsers only directly support GET requests. For this example, we assume that you have installed a REST client browser plugin. Chrome and Firefox both support open source Rest Client plugins that allow for the invocation of REST APIs from the browser.

Procedure

1. Enter the following URL to authenticate to the Cúram server with the j_username and j_password parameters: `https://<host>:<port>/Rest/j_security_check?j_username=<username>&j_password=<password`
2. Select the POST method of the `https://<host>:<port>/Rest/v1/notes` URL.

Option	Description
https://<host>:<port>/Rest	The context path of the URL where you can access the REST APIs.
v1	The version of the REST API that you want to invoke.
notes	The resource you want to access.

3. Set the following required request headers:
   - **Referer**
     curam://foundational.app
   - **Content-Type**
     application/json
4. Add a JSON representation of the note to the request body. For example:

```
{
  "outcome_plan_id": "106",
  "username": "planner",
  "description": "Some text",
  "title": "Some text",
  "status": {
    "value": "RST1"
  }
}
```

5. Submit the POST request.

Results

If the request is successful, you see a HTTP 201 Created response code, indicating the successful creation of the Note.

The HTTP response will contain no body content, however the response header will include a Location property, referencing the URL of the newly created note resource. For example:

```
Location: https://<host>:<port>/Rest/v1/notes/12345
```

This URL follows the same format as the POST URL, but with the additional /{note_id} path parameter added to the end. In the above example, 12345 is the unique identifier of the newly created note.

If the request is unsuccessful or if you are not logged into the Cúram server, you see a HTTP error response code.

.

Making a PUT request

In this example, you modify a Note object by calling the PUT method on the Notes resource.

Before you begin

A PUT request cannot be made using a web browser, as web browsers only directly support GET requests. For this example, it is assumed that you are using a REST client browser plugin. Chrome and Firefox both support open source Rest Client plugins that allow for the invocation of REST APIs from the browser.

Procedure

1. Enter the following URL to authenticate to the Cúram server with the j_username and j_password parameters: `https://<host>:<port>/Rest/j_security_check?j_username=<username>&j_password=<password`
2. Select the PUT method of the `https://<host>:<port>/Rest/v1/notes/12345` URL.

Option	Description
https://<host>:<port>/Rest	The context path of the URL where you can access the REST APIs.
v1	The version of the REST API that you want to invoke.
notes	The resource you want to access.
12345	The unique identifier of the note, passed as a path parameter.

3. Set the following required request headers:
   - **Referer**
     curam://foundational.app
   - **Content-Type**
     application/json

4. Add a JSON representation of the existing note, including the modified property values, to the request body. For example:

```
{
  "note_id": "12345"
  "outcome_plan_id": "106",
  "username": "planner",
  "description": "Some updated text",
  "title": "Some updated text",
  "status": {
    "value": "RST1"
  }
}
```

   In the example, a note_id parameter must be included and must match the note_id path parameter included in the URL. If it is not, it will cause an error. The title and description property values have been updated, but all other properties remain at their original values.
5. Submit the PUT request.

Results

If the request is successful, you see a HTTP 200 OK status code.

The HTTP response contains the modified note in the body content. For example:

```
{
  "outcome_plan_id" : "106",
  "username" : "planner",
  "creationDate" : "2015-06-11T17:41:21.000+0000",
  "description" : "Some text",
  "title" : "Some text",
  "note_id" : "12345",
  "userFullName" : "Sarah Brown",
  "status" : {
    "tableName" : "RecordStatus",
    "value" : "RST1",
    "description" : "Active",
    "parentCodeTable" : null
  }
}
```

The response return contains all the properties of the Note representation even though you did not specify all properties.

If the request is unsuccessful or if you are not logged into the Cúram server, you will see a HTTP error response code.

.

Making a DELETE request

In this example, you delete a Note object by calling the DELETE method on the Notes resource.

Before you begin

You cannot make a DELETE request by using a web browser, as web browsers only directly support GET requests. For this example, we assume that you have installed a REST client browser plugin. Chrome and Firefox both support open source Rest Client plugins that allow for the invocation of REST APIs from the browser.

Procedure

1. Enter the following URL to authenticate to the Cúram server with the j_username and j_password parameters: `https://<host>:<port>/Rest/j_security_check?j_username=<username>&j_password=<password`
2. Select the DELETE method of the `https://<host>:<port>/Rest/v1/notes/12345` URL.

Option	Description
https://<host>:<port>/Rest	The context path of the URL where you can access the REST APIs.
v1	The version of the REST API that you want to invoke.
notes	The resource you want to access.
12345	The unique identifier of the note, passed as a path parameter.

3. Set the following required request headers:
   - **Referer**
     curam://foundational.app
   - **Content-Type**
     application/json
4. Submit the DELETE request.

Results

If the request is successful, you see a HTTP 204 No Content status code, with no content in the response body.

If the request is unsuccessful or if you are not logged into the Cúram server, you see a HTTP error response code.

Cúram REST API security

When you access any REST API resource in Cúram, you must be an authenticated user with valid authorization permissions for the relevant resource methods.

Authentication

Cúram REST APIs are subject to the security controls that are implemented in the Cúram application.

Requesting a Cúram REST API resource before authentication, or without valid credentials, always results in an HTTP response status code of 401 Unauthorized.

To programmatically authenticate to the Cúram server, you can use the j_security_check `POST` URL:

- j_username – The user name.
- j_password – The password.

https://<host>:<port>/Rest/j_security_check?
j_username=<username>&j_password=<password>

The HTTP Response Status Code for a successful authentication is `200 OK`.

The HTTP Response Status Code for an unsuccessful authentication is `401 Unauthorized`.

Cookies

After successful authentication, a number of Cookies are set in the HTTP session to represent the security credentials for the user. The following Cookies, and their correct values, must be included in subsequent requests to remain authenticated:

- JSESSIONID
- LTPA2 (Required for IBM® WebSphere® Application Server only)
- _WL_AUTHCOOKIE_JSESSIONID (Required for Oracle WebLogic Server only)

Authorization

The authenticated user must have sufficient authorization permissions to access the API resource. If a user does not have permission, a 403 Forbidden HTTP response code is returned.

Ensure that you remove Security IDentifiers (SIDs) from the database for any unused REST API functions to greater secure what is available to be accessed by users.

Timeouts

After a certain period, your session will timeout and your user credentials are revoked and no longer valid. This timeout period is the same as for the Cúram web application, usually defaulting to 30 minutes. When the user session times out, subsequent API resource requests result in a 401 Unauthorized HTTP status code.

To start a new session the user or application must reauthenticate, and update the values of the persisted Cookies.

Log out

To log out a user and invalidate the session, send an HTTP `POST` request to the log out URL *`https://<host>:<port>/Rest/logout`*.

The HTTP Response Status Code for a successful logout request is `200 OK`.

Cross-Site Request Forgery (CSRF) protection for RESTful web services

RESTful web services use a combination of mechanisms to protect against Cross-Site Request Forgery (CSRF) attacks.

For more information about CSRF, see the *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet* related link.

- **CSRF and Cúram**
  Cúram RESTful web services use the HTTP referrer header to protect against CSRF attacks. For information about how to configure the referrer header mechanism, see the *Cúram REST configuration properties* related link.
  Token-based protection adds an extra layer of security. Cúram REST infrastructure supports token-based CSRF protection for all REST operations, that is, `GET`, `POST`, `PUT`, and `DELETE`. By default, the token-based CSRF protection mechanism is disabled. For more information about enabling token-based CSRF protection, see the *Enabling token-based Cross-Site Request Forgery (CSRF) protection* related link.
- **Reducing the risk of a CSRF attack**
  By default, REST token-based CSRF protection is disabled. When token-based protection is disabled, the CSRF protection mechanism checks only whether domains are permitted. Enabling token-based protection makes this check stronger. If token-based protection is enabled, then both the domain and its subdomains are checked to identify if the domain and its subdomains are permitted.
  If you enabled token protection and you require subdomain access, ensure that you add the subdomains to the list of trusted host domains in the system property `curam.rest.refererDomains`.

Note: When token protection is enabled and a request comes from a host domain or subdomain that is not in the trusted host domains list, the REST request is blocked. Compromised subdomains make CSRF attacks easier within the parent domain. Therefore, take the necessary steps to protect the integrity of your registered domains and subdomains.

Related concepts

Enabling token-based Cross-Site Request Forgery (CSRF) protection on page 20
By default, token-based Cross-Site Request Forgery (CSRF) protection is disabled.

Related reference

REST configuration properties on page 71
The Cúram REST infrastructure uses five properties. You enable the properties in the `Application.prx` file for your custom component or by using the Cúram administration console.

Related information

Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet

Integrating token-based Cross-Site Request Forgery (CSRF) protection

To integrate the client with token-based Cross-Site Request Forgery (CSRF) protection, refer to the following process.

For more information about enabling token-based CSRF protection, see the *Enabling token-based Cross-Site Request Forgery (CSRF) protection* related link.

1. Before token acquisition can begin, the user must authenticate with the server. They log in by using your normal authentication process.

2. Acquire a new CSRF token from the server by calling the REST endpoint *baseURL/v1/csrf/tokens*. The server generates a new CSRF token and sends the token to the client in a custom HTTP response header with the name X-IBM-SPM-CSRF.
3. The client must retrieve the CSRF token from the custom header and store it. The CSRF token is needed for any subsequent REST API calls.
4. The client must send the CSRF token with every API request. The token is sent by a custom request HTTP header with the name X-IBM-SPM-CSRF.
5. When the server receives a client request, the CSRF token is removed from the request header and is validated by the server. If the CSRF token is valid, the request is marked as valid and the request continues. If the CSRF token is invalid, the request is blocked and a 403 forbidden HTTP status code is returned. The following example illustrates the JSON response object with an error message and error code:

```
{
 "errors": [
  {
"code": -150220,
"message": "The request is forbidden as the specified token is not allowed",
"level": "error"
   }
 ]
}
```

8.2.0.0 For more information about HTTP status codes and internal error codes, see Domain API error codes on page 39.

6. The token is valid for the duration of the session. If the session is invalidated for any reason, the client must reacquire a new CSRF token before it makes any new API calls. The client must update its local store with the new token.

Enabling token-based Cross-Site Request Forgery (CSRF) protection

By default, token-based Cross-Site Request Forgery (CSRF) protection is disabled.

Note: Enabling token-based CSRF protection changes the behavior of the allowed referrer domains. Before you enable token-based CSRF protection, see the *Cross-Site Request Forgery (CSRF) protection* and the *Integrating token-based Cross-Site Request Forgery (CSRF) protection* related links.

When you enable token-based CSRF protection, it affects existing client applications. The following steps outline how you can enable token-based CSRF protection in the Cúram system administration application:

1. Log in to Cúram as a system administrator.
2. Select **System Configurations** > **Shortcuts** > **Application Data**.
3. Type `enable.rest.csrf.validation` in the **Name** field and click **Search**.
4. Select **...** > **Edit Value**.
5. Set the value to **TRUE** and click **Save** to save your changes.
6. Click **Publish** for your changes to take effect.

Related concepts

Integrating token-based Cross-Site Request Forgery (CSRF) protection on page 19
To integrate the client with token-based Cross-Site Request Forgery (CSRF) protection, refer to the following process.

Connecting to a Curam REST API using Swift for Apple iOS

Connecting programmatically to a Cúram REST API resource depends on the programming language used. The following is an example using the Apple Swift Language for iOS.

Before you begin

You must have a MacBook with Xcode installed, and an existing project created.

Procedure

1. Create a LoginService.swift class file in your project to handle the Cúram authentication code.
2. Define the following variables in the LoginService.swift class:

```
var username = ""
var password = ""
var data: NSMutableData = NSMutableData()

// Used for Weblogic
var WL_JSESSION_ID: String  = ""
var JSESSION_ID: String = ""

// Used for Websphere
struct LTPAToken {
  static var ltpaToken2:String = ""
}
```

3. Define the invokeLoginService() function, which will eventually be called to initiate a login request. The function takes three parameters:
   - username

     The user name to log in.
   - password

     The password to log in.
   - loadTokenUrl

The full qualified URL for the login, that is, *`https://host:port/Rest/j_security_check`*

```
func invokeLoginService(username: String, password: String, loadTokenUrl: String)
    {
        let nsUrl = NSURL(string: loadTokenUrl as String );
        self.username = username
        self.password  = password
        NSLog(loadTokenUrl)
        let request = NSMutableURLRequest(URL:nsUrl!);
        request.HTTPMethod = "POST";
        request.HTTPShouldHandleCookies = true
        request.setValue("application/x-www-form-urlencoded", forHTTPHeaderField:
 "Content-Type")
        //set Referer for CSRF
        request.setValue("curam://foundational.app", forHTTPHeaderField: "Referer")
        let postString = "j_username=" + username + "&j_password=" + password ;
        request.HTTPBody = postString.dataUsingEncoding(NSUTF8StringEncoding);
        //invoke login request
        var connection: NSURLConnection = NSURLConnection(request: request,
 delegate: self, startImmediately: true)!
        connection.start()
    }
```

The delegate argument in the NSURLConnection object defines that the LoginService (self) class should handle the response from the login request. Next, define the following connection functions to support this.

4. Define a connection function to handle when the connection request is completed. This function sets the necessary cookies if the request was successful.

```
func connection(connection:NSURLConnection, didReceiveResponse response:
 NSURLResponse) {
        let status = (response as? NSHTTPURLResponse)?.statusCode ?? -1
        NSLog("status code is \(status)")

        if ( status == 200 )
        {
            //200 OK authentication successful, fetch the LTPAToken2 from response
 and set for future requests
            self.data = NSMutableData()
            let cookieStorage = NSHTTPCookieStorage.sharedHTTPCookieStorage()
            let allCookies = cookieStorage.cookiesForURL(response.URL!) as!
 [NSHTTPCookie]

            var cookieProperties = [String: AnyObject]()
            for cookie in allCookies {
                cookieProperties[NSHTTPCookieName] = cookie.name
                cookieProperties[NSHTTPCookieValue] = cookie.value
                println("name: \(cookie.name) value: \(cookie.value)")
                if ( cookie.name == "LtpaToken2" )
                {
                     LTPAToken.ltpaToken2 = cookie.value!
                }
            }
            let success = [ 200,  "Success"]
            return
        }

        else if ( status == ErrorCodeConstants.errAuthenticationFailure )
        {
            //authentication error
            let error = [ 401,  "LoginErrorMsg3"]
            return
        }
        else
        {
            let error = [ 600,  "LoginErrorMsg2"]
            return
        }
}
```

5. Define a connection function to handle the data that is returned by the request.

```
    func connection(connection: NSURLConnection, didReceiveData connectionData:
 NSData) {
        // Append the received chunk of data to our data object
        self.data.appendData(connectionData)
    }
```

6. Define a connection function to handle an unsuccessful response.

```
func connection(connection: NSURLConnection, didFailWithError connectionData:
 NSError) {
        // Append the received chunk of data to our data object
        NSLog("didFailWithError ---> \(connectionData)")
        return
    }
```

7. Use the newly created LoginService.swift class within your project to initiate authentication. For example, from a view controller, use the following:

```
var username = "planner"
var password = "password"
var loginURL = "https://host:port/Rest/j_security_check
let loginService = LoginService()
loginService.invokeLoginService (username, password, loginURL)
```

8. In your project, use the following code to invoke the GET method of the Cúram persons REST API resource. This code uses the previously defined loginService variable.

```
       let url  = NSURL(string: curamServerURI+"/v1/persons/101")
       let request = NSMutableURLRequest(URL:url!);
       request.HTTPMethod = "GET"
       let config = NSURLSessionConfiguration.defaultSessionConfiguration()
       //set the headers , this can be moved in a common method, when there are
more apis to invoke
       var xHTTPAdditionalHeaders: [NSObject : AnyObject] = ["Content-Type":
"application/json", "Accept": "application/json"]
       //set the headers for LTPAToken if it is for WAS else for WLS , you need to
set the _WL_AUTHCOOKIE_JSESSIONID & JSESSIONID
       xHTTPAdditionalHeaders["Cookie"] =
"LtpaToken2="+LoginService.LTPAToken.ltpaToken2+";"
       //set Referer for CSRF
       xHTTPAdditionalHeaders["Referer"]  =  "curam://foundational.app"
       config.HTTPAdditionalHeaders = xHTTPAdditionalHeaders

       let session = NSURLSession(configuration: config)
       var err: NSError?
       let task : NSURLSessionDataTask = session.dataTaskWithRequest(request,
completionHandler: {(data, response, error) in
           let status = (response as? NSHTTPURLResponse)?.statusCode ?? -1
           self.data = NSMutableData()
           if (status == 200)
           {
               //200 OK..request successful .... process response
               println("Response: \(response)")
               var strData = NSString(data: data, encoding: NSUTF8StringEncoding)
               println("Body: \(strData)")
               var err: NSError?
               if let jsonResult = NSJSONSerialization.JSONObjectWithData(data,
options: NSJSONReadingOptions.MutableContainers, error: &err) as? NSDictionary {
                   if(err != nil) {
                       // If there is an error parsing JSON, print it to the
console
                       println("JSON Error \(err!.localizedDescription)")
                   }
                   if let results: NSArray = jsonResult["relationships"] as?
NSArray {
                       dispatch_async(dispatch_get_main_queue(), {
                           println("Person Relationship data :\(results)")
                       })
                   }
               }
           }
           else
           {
               let error = [ 600,  "Request failed"]
               println("Request failed Error \(self.data)")

           }
       })
       //start the service request
       task.resume()
```

Cúram REST API methods

Cúram REST APIs support GET, POST, PUT, and DELETE methods on resources. The GET method is used to read data from Cúram. The POST method is used to create a resource. The PUT method is used to modify a resource, and the DELETE method is used to delete a resource.

Refer to the following example to see how methods operate on a REST resource that allows for the creation, modification, and deletion of notes.

In the example, the note resource is represented by `/notes`. There are two resource paths required:

- The collection resource, used to retrieve all notes and create new notes, `/notes`
- The member resource, used to operating on a single, specific note, `/notes/{note_id}`

Each of these resources supports a number of methods and the following table defines how you might use the methods for the two notes resource paths:

Table 1:

Method	Resource	Notes	Successful HTTP Response Code
GET	/notes/	Gets all notes in the system.	200
POST	/notes/	Creates a new note.	201
PUT	/notes/{note_id}	Updates an existing note.	200
GET	/notes/{note_id}	Returns a specific note.	200
DELETE	/notes/{note_id}	Deletes a specific note.	204

A note resource returns a JSON representation of a note. For example:

```
{
  "note_id": "1234"
  "text": "A new note!!"
}
```

GET Collection Resource

The GET /notes method will return a list of notes, which is represented as an array of notes, where the array is identified by the data property in the JSON representation:

```
{ data : [{
  "note_id": "1234"
  "text": "A new note!!"
},
{
  "note_id": "1235"
  "text": "Another note!!"
}
]}
```

POST Collection Resource

You use the POST /notes method to create a new note. A POST request is made to this resource, with a request body containing representation of the new Note, as follows:

```
{
  "text": "A new note!!"
}
```

No note_id property is passed as part of the representation, as Cúram automatically generates the unique identifier for the note.

The result of this request is a HTTP 201 Created response status code. The response body is empty. The response header contains a location entry that details the URL for the newly requested resource, for example `"location": https://host:port/Rest/v1/notes/1234`.

GET Member Resource

If 1234 is the unique identifier for the newly created resource in the example, you can make a GET request using the /notes/{note_id} member resource URL to retrieve the representation of the Note. For example, https://host:port/Rest/v1/notes/1234. This returns the JSON representation of a single note in the response body.

PUT Member Resource

To modify a Note, use the PUT /notes/{node_id} method. A PUT request is made to this resource (for example, /notes/1234), with a request body containing the representation of the Note to be modified:

```
{
  "note_id": "1234"
  "text": "Updated text"
}
```

Both the note_id and text properties are included in the request body, and the note_id must match the value in the resource URL, that is, 1234 in this example. The result of this request is an HTTP 200 OK response status code and the response body contains a representation of the resource.

The response body returns the modified resource to allow for properties that were modified by Cúram to be updated, for example a version number property that is used for optimistic locking.

DELETE Member Resource

To delete a note, the DELETE /notes/{note_id} method should be used. A DELETE request is made to this resource, with an empty request body. For example, /notes/1234.The result of this request is an HTTP 204 No Content response status code.

Request Headers

When making GET, PUT, POST and DELETE requests for resources on a REST API, a number of request headers can be included to control the response.

- **Content-Type**
  Required for all requests and usually set to application/json. The exception is when requesting binary data, in this instance it should be set to */*.
- **Accept-Language**
  Optional and should be set to change the response content language. Setting an accept-language header only applies to the current request.
  HTTP supports multiple values to be set for the accept-language header, however only the highest priority value is used and all other languages are ignored. It is recommended to set only one locale when using this header property. For example, the following inputs will always result in the de locale being used:
  - Accept-Language: de
  - Accept-Language: de, en-gb
  - Accept-Language: en-gb;q=0.8, de;q=0.9, en;q=0.7

- **Referer**
  Required for all HTTP requests that might modify the state of the server, that is, PUT, POST and DELETE. The value of the Referer header is checked to see whether it meets any of the following conditions:
  - The URI Scheme name exactly matches 'curam' for example, the following Referer URI is valid: *curam://foundational.app*. However, the following does not match *other://somedomain.com*. Using "curam" URI scheme name is useful for non-browser HTTP clients like mobile apps.
  - The scheme is HTTP or HTTPS and the authority (domain) matches *localhost* or the Cúram configured domain.

  If no Referer header is set, a 403 Forbidden HTTP status code response is returned.

Related information

403 Forbidden HTTP status code response on page 37

Response headers

The following response headers are always returned:

- **Cache-Control**
  The period that the response content is cached for. The default value is `private, must-revalidate, max-age=0`, that indicates that the response should never be cached.
- **Content-Length**
  The size of the response object.
- **Content-Type**
  The content type, including the character set. This will usually be *application/json;charset=utf-8*, except for binary content where it will represent the correct media type or application/octet if no media type can be determined.
- **Location**
  The URL of the newly created resource is returned as part of a POST method.

Optional and mandatory properties

When making a PUT or a POST request, the properties of the JSON content in the request body may be optional or mandatory. Mandatory attributes must always be specified and will result in an error if not included.

Where a property is optional and no value is to be specified, this can generally be specified in one of three ways for the POST and PUT methods:

- As an empty string, for example, `{“name”:””}`
- Not specified at all, for example: `{}`
- As null, for example: `{“name”:null}`

Where no value is specified, a default value is assigned by Cúram and on a subsequent GET method request, the default value for the property will always be specified. For example, a Note contains four properties:

- note_id: The note ID, this is required

- text: The note text, this is required
- username: The username of the user that is creating the note, this is optional
- highPriority: Boolean indicating if this is a high priority note, this is optional

A PUT request is set with the following representation for the Note:

```
{"note_id":"1234","text":"Some
        updated text"}
```

The username and highPriority properties, which are optional, are not specified. The resulting representation for a Note would now look like:

```
{"note_id":"1234","text":"Some
        updated text","username":"","highPriority":false}
```

Where username has now defaulted to an empty string and highPriority has defaulted to false.

Related concepts

Cúram REST data types on page 29
A Cúram REST API resource request returns or accepts data in JSON format, which represents the various data types that are supported by Cúram.

System-generated, non-editable, and read-only properties

System-generated, non-editable, and read-only are properties in a JSON representation that are treated differently depending on the method used. They can be identified using the following key words specified in the documentation field for a property in the Cúram-generated Swagger document:

Read-only

A property that is returned as part of the resource representation for a GET method, but can never be modified or created. A read-only property is ignored by the POST and PUT methods.

Non-editable

A property that can be set only by a POST method. The value of such a property is ignored by a PUT method and cannot be modified after creation.

System-generated

A property that cannot be set by a POST method during creation, but that can be modified by a PUT method. The value of such a property is ignored by a POST method.

API Versions

The version number of the REST API resources that you are using is included in the URL.

For example, *`https://<host>:<port>/Rest/v1/persons/101`*, where v1 is the version number and usually the first path parameter after the context path, *`https://<host>:<port>/Rest`*.

A new version of an API is created where changes are made to the resources in the API that impact users of the API. For example, the addition of new optional properties to a resource does not mean that a new version is created. However, removing properties from the resource representation does create a new version.

Cúram REST data types

A Cúram REST API resource request returns or accepts data in JSON format, which represents the various data types that are supported by Cúram.

The following table outlines the primitive data types that are supported by Cúram, how they are represented. and their default values.

Table 2:

Cúram Type	Java Type	JSON Representation	Default Value	Notes
SVR_CHAR	char	string	A space	
SVR_STRING	String	string	Empty string	
SVR_BOOLEAN	boolean	boolean	false	
SVR_INT8	byte	integer	0	
SVR_INT16	short	integer	0	
SVR_INT32	int	integer	0	
SVR_INT64	long	string	"0"	Represented as a string to cater for numbers that exceed JSONs supported number values.
SVR_FLOAT	float	float	0.0	
SVR_DOUBLE	double	double	0.0	
SVR_DATE	n/a	string : ISO 8601 format	Null	A date specified in the ISO 8601 format, for example 2015-12-01
SVR_DATETIME	n/a	string: ISO 8601 format	Null	A date time specified in the ISO 8601 format, for example 1938-04-26T23:00:00.000+0000
SVR_CODETABLE	n/a	{value:"", description:"", tableName:"", parentCodeTable:""}	Null	Represented as an object.
SVR_BLOB	n/a			
FREQUENCY_PATTERN	n/a	{value:"", description:""}	Null	Represented as a JSON object.

Cúram Type	Java Type	JSON Representation	Default Value	Notes
8.2.0.0 INFOMATIONAL_MESSAGE	String	{ "code": 400 "message": "this is the localized error message" "level": "warning" }	Null	Represented as a JSON object. For more information about HTTP status codes and internal error codes, see Domain API error codes on page 39.

Date and date time

Date and Date Time formats are specified by using the ISO 8601 format.

For example:

YYYY-MM-DDThh:mm:ss.sss[+hhhh]

Dates are timezone independent and do not specify the time portion of a date. If the time portion is specified, it is ignored. Example accepted inputs:

- 2015-01-24
- 2015-01-24T23:00:00.000+0000

T23:00.00+00 is ignored.

Example output:

- 2015-01-24

Date Time values are timezone-dependent and for PUT and POST methods, the time should be specified in the UTC timezone, or the timezone offset (hours from UTC) should be included. Example accepted inputs:

- 2015-01-24T16:55:00.000+0000
- 2015-01-24T18:55:00.000+02:00
- 2015-01-24T16:55:00.000
- 2015-01-24T16:55:00.000Z

Example output:

- 2015-01-24T16:55:00.000+0000

Note: The value of the milliseconds portion of the ISO 8601 format is ignored by Cúram. Any millisecond values are always rounded down to 000.

Code tables

A code table is a Cúram specific datatype that is used for collections of commonly used constants. They allow for a locale independent way of encoding values. For example, a status code table might contain the values 'Open' and 'Closed'. In French these values would be 'Ouvert' and

'Ferme'. Rather than have the client and server try to interpret these translated values, the values are encoded, for example, the codes 'O' and 'C' might be used to represent the status.

For a Cúram REST API resource representation, a code table is represented as a JSON object containing the following four properties:

- value: The value of the code table entry.
- description: The localized description of the code table entry.
- tableName: The name of the table this code table entry belongs to.
- parentCodeTable: The name of the parent code table. This is only used where the code is part of a code table hierarchy and otherwise is null or not specified.

An example of a code table JSON object

```
"addressType": {
     "tableName": "AddressType",
     "value": "AT1",
     "description": "Private",
     "parentCodeTable": null
}
```

For PUT and POST methods, only the value property is required to be provided. All other properties will be ignored. A shorthand way of specifying the code table value is also available, where the content of the value property can be passed up directly. The following inputs are accepted for a code table value in a PUT and POST method:

```
"addressType": {
     "tableName": "AddressType",
     "value": "AT1",
     "description": "Private",
     "parentCodeTable": null}
```

```
"addressType": {"value": "AT1"
}
```

```
"addressType": "AT1"
```

The Cúram generated Swagger document contains a reference to the code table that represents the full list of entries that can be specified for a particular field. For example, a property that is related to the AddressType code table would contain the following documentation:

“The value of this property must be an item from the AddressType code table. See `/codetables/AddressType`”

This documentation refers to the `/codetables/{table_name}` resource, which supports a GET method that can be called to retrieve the code table entries for the specified table name. This is helpful where a user interface needs to display the full list of available code table entries in a drop down select list.

The result of a GET request for the `/codetables` resource contains the full list of localized entries, along with the default code table entry. For example:

```
{{
    "tableName" : "AddressType",
    "defaultValue" : "AT1",
    "codeItems" : [ {
        "value" : "AT1",
        "description" : "Business",
        "sortOrder" : 0
    }, {
        "value" : "AT2",
        "description" : "Institutional",
        "sortOrder" : 0
    }, {
        "value" : "AT3",
        "description" : "Mailing",
        "sortOrder" : 0
     } ]}
```

Code table hierarchies

A code table can be part of a hierarchy, which is a set of code tables where selecting an entry in one list results in a subset of entries being displayed in the related child list.

The SpecialCautionCategory code table is an example of such a hierarchy. It has two levels in the hierarchy, represented by the SpecialCautionCategory (parent) and SpecialCautionType (child) code tables.

SpecialCautionCategory, for example, has two entries: Behaviour Alert and Safety Alert

If Behaviour Alert is selected, SpecialCautionType displays: Difficulties at School and Runaway Risk.

If Safety Alert is selected, SpecialCautionType displays Risk of Falls and Violent Offender.

The parentCodeTable property in the JSON representation of a code table entry indicates if an entry is part of a hierarchy and which table is the parent.

The result of a GET request to the `/codetables/{table_name}` resource, where the code table is the parent in a hierarchy, contains the full list of localized entries, including details of the child entries for each entry in the code table. For example:

For example, to represent a Special Caution the JSON representation would look like:

```
{
"specialCautionCategory": {
"tableName": "SpecialCautionCategory",
"value": "SCC02",
"description": "Safety Alert",
"parentCodeTable": null },
"specialCautionType": {
"tableName": "SpecialCautionType",
"value": "SCSC10",
"description": "Violent Offender History",
"parentCodeTable": "SpecialCautionCategory" },
}
```

Two properties are defined, one for each entry in the code table hierarchy. The child item, specialCautionType, has a value set for the parentCodeTable property, SpecialCautionCategory. This is the name of the parent code table. The parent item, specialCautionCategory, has no value set for the parentCodeTable property (that is, null).

Like code tables, the /codetables/{table_name} resource can be used to retrieve the full information associated with the code table hierarchy. The {table_name} specified should be the parent code table name and this is document in the Curam generated Swagger document. The following is an example result of the /codetables/SpecialCautionCategory resource:

```
{
  "tableName" : "SpecialCautionCategory",
  "defaultValue" : "",
  "codeItems" : [ {
    "value" : "SCC03",
    "description" : "Behavioural Alert",
    "sortOrder" : 0,
    "childCodeTable" : {
      "tableName" : "SpecialCautionType",
      "defaultValue" : "SCSC11",
      "codeItems" : [ {
        "value" : "SCSC12",
        "description" : "Difficulties at School",
        "sortOrder" : 0
      }, {
        "value" : "SCSC11",
        "description" : "Runaway Risk",
        "sortOrder" : 0
      } ]
    }
  }, {
    "value" : "SCC02",
    "description" : "Safety Alert",
    "sortOrder" : 0,
    "childCodeTable" : {
      "tableName" : "SpecialCautionType",
      "defaultValue" : "SCSC08",
      "codeItems" : [ {
        "value" : "SCSC09",
        "description" : "Risk of Falls",
        "sortOrder" : 0
      }, {
        "value" : "SCSC10",
        "description" : "Violent Offender History",
        "sortOrder" : 0
      } ]
    }
  } ]
}
```

Frequency patterns

A frequency pattern is a Cúram specific datatype that is used to represent the frequency of an occurrence, for example, how often a meeting should be scheduled for. A frequency pattern is a nine digit, non-human readable, number.

For a Cúram REST API resource representation, a frequency pattern is represented as a JSON object containing the following two properties:

- value: The nine digit representation of a frequency pattern
- description: The localized display text for a frequency pattern

The following is an example of a JSON object frequency pattern:

```
"frequency": {
           "value": "100101600",
           "description": "Recur every 1 week(s) on Friday"
}
```

For PUT and POST methods, only the value property is required to be provided. The description property is ignored if specified.

A shorthand way of specifying the frequency pattern value is also available, where the content of the value property can be passed up directly. The following inputs are accepted for a frequency pattern value in a PUT and POST method:

```
"frequency": {
     "value": "100101600","description": "Recur
      every 1 week(s) on Friday"
}
```

```
"frequency": {
     "value": "100101600"
}
```

```
"frequency": "100101600"
```

Binary data

A resource is typically represented by JSON data. However, some resource methods allow for sending and retrieving binary data.

When a GET method returns binary data, the Swagger document that is generated by Cúram specifies the Response Content type as anything other than *`application/json`*. This response indicates that binary content and not JSON content is provided in the response.

The response body of the requested resource method contains the binary content and the response headers indicate the file size (Content-Length) and the file type (Content-Type).

Where binary content is expected as input on a POST or a PUT method, the documented input type must be specified correctly, that is, not *`application/json`*.

For a POST or a PUT method, the request body contains the binary content to be provided.

Not all open source or commercial Swagger tools support the handling of binary content and one alternative is to use cURL. cURL is a command-line utility for transferring data by using various protocols, including HTTP.

Example CURL commands for sending binary content:

```
curl -v -k -c ~/.cookies -X POST 'https://server:port/Rest/j_security_check?
j_username=username&j_password=password'

curl -v -k -b ~/.cookies -X POST -H "Content-Type: application/octet-stream" -H
 "Referer: curam://foundational.app" --data-binary @testFile.jpg 'https://server:port/
Rest/v1/files/'
```

Where:

- The first **`curl`** command logs in with the user name and password and creates a file of type *`.cookies`* that contains the necessary authentication information.
- In the second **`curl`** command, *`@testFile.jpg`* is the binary file to be uploaded and *`\https://server:port/Rest/v1/files/`* is the resource URL handling POST method for binary content

Lists and nested structures

A resource representation, in the JSON format, is a complex JSON object. Lists of items and nested data structures are represented as JSON arrays and nested JSON objects.

A resource that returns a list of items, represents those items in the following standard format:

```
{"data": [{…},{…}]}
```

The data property is an array of JSON objects, where each object represents an item in the list.

A resource may also represent some of its data in a nested JSON object. For example:

```
{
      "outcome_plan_id": "101",
      "assignedTo": {
     "fullName": "Becky Fernes",
     "concern_role_id": "101",
     "username": "bfernes"
      }
}
```

The assignedTo property in the example is a nested object containing additional properties.

A nested object may also be an array of objects, for example,

```
{
      "outcome_plan_id": "101",
      "concerning": [{"fullName": "Becky Fernes","concernRoleID": "101"},
{"fullName": "Mary Fernes","concernRoleID": "102"}]
}
```

The concerning property in the example is an array of objects, that define the full name and concern role ID of related participants.

There is no restriction on the level of nesting of of arrays and objects in resource representations, however as best practice no more than three levels should exist in a resource representation.

Informational Message Pattern

An informational message pattern is a Cúram specific datatype that is used to represent a number of warnings, or errors. For example, the messages are collected and shown to user in a localized format.

Common usage patterns

You can use the following common patterns when you access Cúram REST APIs.

File download

A resource GET method that returns binary content, in addition to the JSON representation of the resource, may include a separate link to the binary content.

For example, a Person resource may include details about the person and a photograph of the person. The JSON representation of this Person resource would look like

```
{
  "concern_role_id" : "101",
  "dateOfBirth" : "1964-09-26",
  "full_name" : "James Smith",
  "photo" : "/vyfcommnhyyccjdy/101",
}
```

In this example, the binary data for the photograph was not included in the JSON representation, instead a separate relative path URL was specified. This URL can be accessed to download the binary content for the photograph using the GET method for the resource URL.

File upload

Uploading files, or binary content, and attaching or associating them with additional data requires a two step process. The first step is to send the binary content to the server, and the second is to associate the file/binary content with a resource.

The following example flow shows the creation of an attachment resource, which represents a file and also contains details on when the file was created and by whom.

First you send the binary content for the file using the request method, *`POST /v1/files`*. The Request Body contains the binary content for the file. The Response from the system is the 201 HTTP status code. The header of the response contains a Location field indicating the relative location of the uploaded file, for example, *`/v1/files/12345`*

Then, you associate the file with the *`/attachment resource`* using *`POST /v1/attachments`*. The request body is as follows:

```
{
     “attachmentName”:”some file”,
     “creationDate”:”2015-12-12”,
     “username”: “someuser”,
       “path”:”/files/12345”
 }
```

The 201 Response code is received. The header of the response contains a Location field indicating the URL of the newly created attachment resource, for example, `/attachments/5678`

When the two steps are successfully completed, the upload is complete.

Troubleshooting REST APIs

Use the following topics to help you troubleshoot problems that you may encounter using Cúram REST APIs.

8.2.0.0 403 Forbidden HTTP status code response

Condition

You receive a 403 Forbidden HTTP status code response when you make request.

Cause

There are three scenarios in where a 403 forbidden HTTP status code is returned. The following list outlines the three scenarios:

- A user does not have sufficient access rights to start the requested resource method.
- A valid Referer header is not specified.
- Enhanced Cross-Site Request Forgery (CSRF) is enabled, but a valid CSRF token is not specified.

Remedy

Procedure

1. To diagnose if the user does not have sufficient access, check if the user's security role and group allow the user to start the requested API resource. If the user does not have sufficient access, then assign the user a security role and group that has the correct privileges.
2. To diagnose if a valid Referer header value was set when the request was sent, for mobile applications, check that the Referer header is set to a URI beginning with `curam://`. For web browsers, check that the Referer header is set to the domain of the server you are accessing, for example, `https://server:port/Rest/<requested_resource_path>`. If the header value is not valid, correct it to resolve the issue. Verify that the *curam.rest.refererDomains* property is set correctly. The Referer header set on a request must be a subdomain of the domains that are specified for the *curam.rest.refererDomains* property. You can use the Cúram administration console to modify the list of domains supported.
3.

To diagnose if an invalid CSRF token is specified or if no token is specified, check whether the following error message is in the response body:

```
{
  "errors": [
   {
 "code": -150220,
 "message": "The request is forbidden as the specified token is not allowed",
 "level": "error"
    }
  ]
}
```

For more information about HTTP status codes and internal error codes, see Domain API error codes on page 39.

4. Check that the HTTP request contains a CSRF token. The token is set as a custom HTTP header. The name of the custom header is X-IBM-SPM-CSRF. If the header is set with a token value, the token might be expired. So, retrieve a new CSRF token by calling the API endpoint `baseURL/v1/csrf/tokens`.
5. Call the API endpoint again by using the new CSRF token. When you make the request, ensure that the new token is passed in the custom header X-IBM-SPM-CSRF.

415 Unsupported media type response

Condition

You receive a 415 Unsupported media type HTTP response when you make request.

Cause

A 415 Unsupported Media Type HTTP status code is returned when a valid Content-Type header has not been specified.

Remedy

Procedure

Ensure that the Content-Type header property is set. In most cases, the Content-Type header property should be set to `application/json`.

Domain APIs

In Cúram, Domain application programming interfaces (APIs) are a set of REST APIs that are designed to support the rise in interoperability prompted by digital transformation initiatives across different industries.

Domain APIs are based on concepts from Domain-Driven Design (DDD). Domain APIs are intended to closely match the business language in the Domain. Domain APIs wrap around the backend complexity to shield the API consumer from being exposed to the Domain API.

Some private APIs are created for a narrow purpose, such as supporting a single front end. In contrast, Domain APIs are designed for reuse. To make Domain APIs more consumable, Domain APIs implement a common approach to field naming and to the sharing of coded data.

Getting started with Domain APIs

Start here if you're new to Domain APIs, or you want further information about Cúram APIs. Otherwise, you can skip directly to the API catalog that lists the Domain APIs.

Domain API error codes

In response to each API call, Cúram returns a message with an appropriate HTTP status code and a more specific error code in the response body.

The following table explains the returned HTTP response status codes.

Table 3: An explanation of the returned HTTP response status codes.

HTTP response status codes	Explanation
`200 OK`	The message is processed successfully.
`400 Bad Request`	While the system was processing the request, an error occurred. The inclusion of certain data in the message caused the error.
`401 Unauthorised`	The caller did not include a valid authentication token.
`403 Forbidden`	The caller does not have sufficient permission to make the request.
`404 Not Found`	The resource does not exist.
`500 Server`	While the system was processing the request, a server-side error occurred.

In addition to the HTTP status code, failure response bodies include fields that provide a more detailed explanation of the error. The following table outlines an explanation of the fields in the failure response bodies.

Table 4: Explanation of the fields in the failure response body.

Fields in the failure response body	Explanation
8.2.0.0 `Code`	The HTTP code or an internal code that represents the exception. Internal codes will have a negative value. If you do not require special handling for internal error codes, by default this code can be interpreted and handled the same as 'HTTP 400 the data request is invalid' The full set of internal code values and their associated message can be obtained from the following property files: • `EJBServer/components/Rest/rest/properties/RestErrorMessages.properties` • `CuramCDEJ/doc/defaultproperties/curam/omega3/i18n/RuntimeMessages.properties`
`Field`	The field is populated where a field API request can be linked to the exception.
`Message`	The detailed failure message that consists of the code and a message string that is separated by a hyphen.
`Message_id`	The field is populated where a unique identifier exists for the exception.
`Level`	The failure level. Error is typically the failure level.
`Stack trace`	The stack trace associated with the error (This may be null if curam.trace is set to trace_off instead of trace_on).

Domain API lists

Some APIs return lists of results to the API caller. You can include canceled records in the results.

Canceled records

By default, when you use GET APIs calls with no parameter only active records are returned in the results. If you must include canceled records in the results, use the include_cancelled query parameter and set the parameter to `true`. The following table outlines the results of applying the include_cancelled query parameter to a GET request.

Table 5: The results of applying the include_cancelled query parameter to a GET request.

API	Results
`GET /v1/openapi/core/persons/12345/cautions`	Only active cautions are returned.
`GET /v1/openapi/core/persons/12345/cautions?include_cancelled=true`	Active and canceled cautions are returned.

Expired records

A period is associated with some records. The period includes a start date and an end date. If the end date is passed, then the record is deemed expired and the record is not included in the list of records.

Securing and enabling the Files API

By default, the file `POST /v1/api/docs/files` API is disabled for security purposes. Before you can enable the Files API and safely upload documents, you must have appropriate file security and validation for any files that are uploaded to your system. If you don't manage these functions elsewhere, such as in content management system or a gateway, then you can use the provided hook point to implement file security and validation.

Before you begin

Note: Enabling the Files `POST /v1/api/docs/files` API with the `curam.rest.docservice.fileupload.enabled` property also enables the Document Service `POST /v1/dbs/files` API that is used for file uploads in the .

For more information about REST APIs in the Cúram Universal Access Responsive Web Application, see the *Universal Access Responsive Web Application Guide*.

About this task

If you don't manage file security and validation for files that are uploaded to your system elsewhere, you can use the provided `RESTFileValidation` hook point. Override the provided dummy implementation with your own custom implementation.

The `RESTFileValidation` interface consists of a single `validateFile()` method that must do all the required virus scans and file checks. The following list outlines the required virus scans and file checks:

- Scanning for viruses.
- Validating the size of the file.
- Validating the file extension.
- Validating that the file name does not contain paths.

The dummy implementation of the interface logs a warning to ensure that this function is implemented.

You can use a Google Guice binding to overwrite the dummy implementation. For more information about using Guice, see the *Persistence Cookbook*.

Procedure

1. If needed, you can use the provided `RESTFileValidation` hook point to implement file security and validation functions as follows:
   a) Use a module class in a custom component. The component extends the `com.google.inject.AbstractModule` class.
   b) Create a custom implementation of the `RESTFileValidation` interface. If there is a problem with a file, ensure that an `AppException` with the appropriate error message

is thrown. The REST infrastructure manages the `AppException`, and returns the error message in the API response body of the JSON and the correct HTTP error code.

c) Bind the new custom implementation to the interface in the `configure()` method. For example, the `configure()` method can include an entry in the format:

```
bind(RESTFileValidation.class).to(RESTFileValidationCustomImpl.class);
```

2. When you are sure that appropriate file security and validations are in place, enable the Files API in the Cúram system administration application:
   a) Log in to Cúram as a system administrator.
   b) Select **System Configurations** > **Shortcuts** > **Application Data** > **Property Administration**.
   c) Type `curam.rest.docservice.fileupload.enabled` in the **Name** field and click **Search**.
   d) Select **...** > **Edit Value**.
   e) Set the value to **TRUE** and click **Save** to save your changes.
   f) Click **Publish** for your changes to take effect.

Exploring API use cases (with real-world examples)

Linking a file to a person is a real-world scenario that demonstrates how you can use a set of Domain APIs to meet a business requirement and create value in an organization. Rather than an exhaustive list that exactly matches your reason for interest in the APIs, the use case shows a potential implementation of the APIs.

Linking a file to a person

When a file, such as a document or a photo, is submitted to an agency you can use the File Links APIs to create a direct link in Cúram to a person or a case. Linking a file to either a person or a case is a way to attach a file as supplemental information. A caseworker or other system users with appropriate security privileges can then access the file.

Business problem

An agency asked citizens to post documents to provide supplemental information. An agency mail room handles the documents that are posted by citizens. The agency also asked the citizens to provide their Social Security Number (SSN) when citizens submitted their documents so that the agency can identify the citizens on the system.

A citizen, Abby White, mailed a copy of her passport with her SSN to the social development agency. A mail room worker, Jack Green, needs to attach the document to Abby's person home page so that Abby's caseworker can review it as part of Abby's case management.

Why integrate Cúram with an external system?

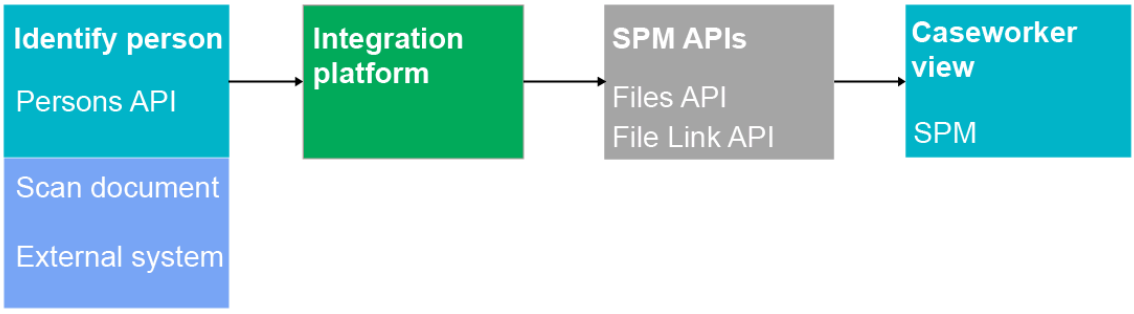
Before the agency uploads the file, or the file location, to Cúram, the agency typically stores documentation on an external document management system or a secure server.

In this scenario, agency mail room worker Jack opens the mail and uses the SSN provided by Abby to identify Abby on the system. Jack uses an external system to scan the document and

the system uploads the file to Cúram. The file is now linked to Abby's person home and the system alerts Abby's caseworker that an attached document requires review. The process flow improves the efficiency of the file linking process and reduces the risk of attaching a document to an incorrect person.

The following diagram shows the end-to-end workflow.

Figure 1: File Link API example: mail room scenario.



1. A citizen, Abby White, posted documents with her SSN to the agency. A mailroom worker, Jack Green, uses the Persons API to find Abby on the system by using Abby's SSN.
2. Jack uses an external system, for example IBM® Datacap, to scan the document. The external system converts the file to an electronic version and then sends the file and Abby's person ID to the Integration Platform layer.
3. The Integration Platform layer receives the data and calls a sequence of APIs in Cúram to upload the file and link the file directly to Abby White's person home.
4. The file is displayed on the **Attachments** tab on Abby's home page so that Abby's caseworker can review the file.

How is person, Abby White, identified in Cúram?

To find a person in Cúram, use the Persons API. You can configure Cúram so that a list of identifications is associated with each person. The system uses identification records to store different forms of participant identification, such as passport numbers and social security numbers (SSNs). Organizations generally use identification records to identify and search for participants. The following table illustrates the mandatory (*) query parameters to include in the Persons API search.

Table 6: Mandatory (*) query parameters to include in the Persons API search.

API attributes	Description
`"identification_type_code": "CT123"`	**Identification Type Code *** - CT123 is the coded value for the identification type “SSN” in Cúram.
`"identification_value": "777-333-444"`	**Identification Value *** - 777-333-444 is the value of Abby White’s SSN stored in Cúram.

```
curl -v -k -b -X -H 'Cookie: LtpaToken2=<value_of_ltpa_token>' \
-H 'Referer: curam://test.app' \
'https://<server>:<port>/Rest/v1/api/core/persons?
identification_value=777-333-444&identification_type_code=CT123'
```

How is the file uploaded to Cúram?

To upload a file to Cúram, use the Files API. This API expects the file to be passed as binary contents in the body of the request. Metadata about the file is included in the request header to be stored with the file in Cúram. The following table illustrates the mandatory attributes (*) to include in the Files API header.

Table 7: The mandatory attributes (*) to include in the Files API header.

API attributes	Description
`"file_name": "img1234.png"`	**File Name *** - img1234.png is the file name after the document is scanned.
`"type_code": "CT2345"`	**Type Code *** - CT2345 is the coded value for the passport document type in Cúram.
`"submitting_application ": "mailroom scan"`	**Submitting Application *** - mail room scan is the portal application that Jack uses to scan the document.

```
curl -v -k -b \
--data-binary @img1234.png
-H 'Content-Type: application/octet-stream' \
-H 'x-ibm-curam-file-metadata: {'file_name':'img1234.png',
 'type_code':'CT2345','submitting_application':'mailroom scan'}' \
-H 'Referer: curam://test.app' \
-X POST 'https://<server>:<port>/Rest/v1/api/docs/files'
```

How is the file linked to the person, Abby White, in Cúram?

To link a file to a person in Cúram, the Person File Link API must receive certain attributes and associated values. The following table highlights the mandatory attributes (*) that are passed in the Person File Link API call.

Table 8: The mandatory attributes (*) that are passed in the Person File Link API call.

API attributes	Description
`{` `"person_id": "123456780001",`	**Person id *** - 123456780001 is Abby White's reference that uniquely identifies Abby White in Cúram.
`"file_id": "222000333000444",`	**Field id *** - 222000333000444 is the reference that uniquely identifies the file in Cúram.
`"description": "Copy of Abby White's passport"` `}`	**Description *** - the description that is associated with the file in the person context.

```
curl -v -k -b -d '{"person_id":"123456780001", "file_id":"222000333000444",
 "description":"Copy of Abby Whites passport"}' \
-H 'Cookie: LtpaToken2=<value_of_ltpa_token>' \
-H 'Content-Type: application/json' \
-H 'Referer: curam://test.app' \
-X POST 'https://<server>:<port>/Rest/v1/api/core/persons/123456780001/file_links'
```

Error handling

For more information about the HTTP status codes that indicate success or failure in API calls, see Domain API error codes on page 39.

Cúram Domain API catalog

The available REST APIs are documented in Swagger so that you can review the API calls before you integrate your application with Cúram. A Swagger spec is available in a running Cúram application and may be viewed using any Swagger UI tool.

The following is a list of the Cúram REST APIs that are available.

Persons API

Use the Persons API to return specific information that is recorded for a person or a prospect person in Cúram.

See the /v1/api/core/persons under the Domain API section in the Swagger spec.

Cautions API

Use the Cautions API to return special caution information that is recorded for a person in Cúram.

See the /v1/api/core/special_cautions under the Domain API section in the Swagger spec.

Person Notes API

Use the Person Notes API to return and view the most recent version of a note that is recorded for a person in Cúram.

See the /v1/api/core/persons/{person_id}/person_notes under the Domain API section in the Swagger spec.

Verifications API

Use the Verifications API to view details for verifications or to return and view all verifications for a specified person or case in Cúram.

See the /v1/api/core/verifications under the Domain API section in the Swagger spec.

Case Notes API

Use the Case Notes API to return and view the most recent version of a note that is recorded on acase in Cúram.

See the /v1/api/core/cases/{case_id}/notes under the Domain API section in the Swagger spec.

Case Overview API

Use the Case Overview API to return unique information that is recorded on a case in Cúram.

Cúram includes various case types. The Case Overview API is designed to return an overview of data in the set of fields that is common to all case types.

See the /v1/api/core/case_overviews under the Domain API section in the Swagger spec.

Files API

Use the Files API to upload, retrieve, or delete a file in Cúram.

Ensure that you use one of the linking APIs to link to the file to upload, retrieve, or delete.

Before you begin: Ensure that you review Securing the Files API.

File Locations API

Use the File Locations API to record a file location when the file is not stored in Cúram.

The File Location API records a link to an external system, such as a document management system, where the file is stored.

Ensure that you then use either the Persons File Links API or the Case File Links API to link the file location to an object.

See the /v1/api/docs/locations under the Domain API section in the Swagger spec.

Person File Links API

Use the Person File Links API to link a specific file to a specific person that is stored in Cúram.

Before you begin: Ensure that the file that you want to link to is uploaded to the system. You can use the Files API to upload the file to the system.

See the /v1/api/core/persons/{person_id}/file_links under the Domain API section in the Swagger spec.

Case File Links API

Use the Case File Links API to link a specific file to a specific case that is stored in Cúram.

Before you begin: Ensure that the file that you want to link to is uploaded to the system. You can use the Files API to upload the file to the system.

See the /v1/api/core/cases/{case_id}/file_links under the Domain API section in the Swagger spec.

1.2 Developing inbound REST APIs

If the existing Cúram REST APIs do not meet all of your requirements, you can create custom inbound REST APIs. For example you can create APIs to integrate mobile applications that connect Cúram with mobile users. For more information about the existing REST API resources, see the related information links.

Before you begin

The Cúram REST API development documentation assumes that you are familiar with REST APIs.

You need access to a Cúram development environment.

Before you start extending the Cúram REST APIs, ensure that you do a gap analysis on the existing REST API resources and methods. Before you start modeling facades for your REST API resources, do a gap analysis of the existing Cúram functions to determine the correct domain definitions to use for your resource representation and to identify any existing API within Cúram that can be used to implement the functions of the new REST API.

Related concepts

Integrating with the Cúram REST API on page 9
Cúram supports the creation of REST APIs and provides a collection of REST API resources. A Cúram REST API is a set of URL resources that you can use to create, read, modify, and delete data in a Cúram system.

Creating a Cúram REST API

Complete the following steps to create a simple Cúram REST API. After you create a custom REST API, you can use it to integrate with other applications, for example, a mobile app.

Related concepts

Integrating with external applications through REST APIs on page 9
You can connect to existing inbound Cúram REST APIs or create your own custom inbound REST APIs. You can also make outbound API requests to integrate with external applications that expose REST APIs.

Cúram REST API design basics

A Cúram REST API is a collection of URL resources that can be used to create, read, and modify data in a Cúram system.

Resources are complete business objects and each resource represents an object in Cúram. Each resource, which is identified by a path, has a set of methods that can act on the resource to create, read, modify, or delete the resource. The resource is represented by a JSON object.

REST is different from the RPC (Remote Procedure Call) style APIs developed within Cúram. RPC APIs usually target specific information or data to be displayed on a particular user screen, and for performing modifications to that information. REST APIs transfer and act on representations of complete business objects.

The design and granularity of REST APIs is important and you must carefully design the REST API before you begin development.

To design a REST API you must identify the major elements of the API, specifically:

- Resources: are identified by using URI path components.
- Methods: define the available operations on a REST resource.
- Representations: are the definition of the structure of the business object that is being represented.

Cúram REST API design principles

When you design custom REST API resources, follow these conventions to ensure good REST API design.

General design conventions

- Use plural nouns and lowercase letters for resource path names, for example, `/notes` for the note resource or `/persons` for the person resource.
- Use snake case for URL query parameters and resource names, for example, *concern_role_id* and *full_name*.
- Representations must remain consistent across operations. The same representations need to be used for GET, PUT, and POST.
- Identifiers must not be in URI form. Do not use HATEOAS style.

Conventions for collection resources and member resources

- A collection resource returns all objects in the collection, for example, `/persons` returns all persons in the system. Typically, collection resources have a required query parameter to reduce the results set, for example, `/persons/full_name=James Smith`.
- The response to a collection request is an object with a property named `data`. This property is an array that holds all of the matching members of the collection.
- A request for a collection where the query parameter does not match any resources returns a successful 200 status code and an empty data array.
- A member resource is an individual item in a collection, for example, `/persons/106`.
- A request for a member resource that does not exist returns a 4xx status code

Cúram REST API resource design example

To illustrate the design of a REST API, the following simple example exposes a note object as a REST resource and allows for the creation, modification, and deletion of notes.

You must decide the path name for the resource. The conventions for REST APIs are to use plurals in the path names, so in this example you use `/notes` to represent the note resource. This requires two resource paths:

- `/notes` The collection resource, used to retrieve all notes and create new notes.
- `/notes/{note_id}` The member resource, used to operate on a single resource.

Cúram REST API method design example

To illustrate the design of a REST API, the following simple example allows for the creation, modification, and deletion of notes. You must define the supported methods. There are 4 possible methods, specifically GET, POST, PUT, and DELETE.

The following table defines the methods to be supported for the two notes resource paths.

Method	Resource	Comment
GET	`/notes/`	A collection resource that returns all the notes in the system
POST	`/notes/`	A collection resource to create a new note
PUT	`/notes/{note_id}`	A member resource to modify a specific note
GET	`/notes/{note_id}`	A member resource to return a specific note
DELETE	`/notes/{note_id}`	A member resource to delete a specific note

Cúram REST API representation example

Shows you different examples of how you can extend the JSON representation for your designs.

You must define the representation of the note object. A resource representation is returned in the JSON format and is best designed in this format. A simple note object is represented with two properties, note_id and text. For example, in JSON this would look like:

```
{
  "note_id": "1234"
  "text": "The note text"
}
```

The following example is a simple representation. However, representations can be complex and contain nested objects, lists of properties, or even lists of nested objects. You can extend the JSON note representation to include an array of users and the date that they modified the note:

```
{
  "note_id": "1234"
  "text": "The note text"
  "modificationHistory": [
      {"username":"sbrowne",
        "modificationDate":"2015-01-01"
      },
      {"username":"asmith",
        "modificationDate":"2015-05-01"
      }
  ]
}
```

You can also include additional information about the resource by using a nested resource. For example, instead of including the modification history as part of the representation, you can separate the information as a nested resource, */notes/{note_id}/modification_history*, where the modification history is represented as follows:

```
{
   "data": [
      {"username":"sbrowne",
        "modificationDate":"2015-01-01"
      },
      {"username":"asmith",
        "modificationDate":"2015-05-01"
      }
  ]
}
```

The */notes/{note_id}/modification_history* resource would have its own set of methods that operate separately on the modification_history data.

Nested resources

Nesting resources provide REST API consumers an easy and efficient way to manage data by allowing the consumer to send and receive only the required object. The nested resource must be a business object, that is, it must still represent a complete business object.

To decide whether a resource should be nested, you must consider the following:

- If the nested array is a stand-alone entity that is referred to outside of the context of the parent resource.

  If the nested array is referenced outside the context of the parent resource, it may be defined as a root resource.
- If the resource is modified frequently.

  If a small part of a resource is frequently modified, each modification invalidates the cache state of the whole resource. In this case, it is good design practice to separate the content into a nested resource. For example, PUT requests on an entire resource that only adds a single item in an array, in cases where the array changes frequently, are wasteful and not a good design.
- If the size of the representation is large or small.

If it is small, perhaps it can be included in the main representation. If it is large, it can dominate the main representation and might be more ideal as a nested resource.

Modeling Cúram REST APIs

Once you complete the design of a Cúram REST API, you must use IBM Rational Software Architect to model the Cúram facades, operations, and structs that implement the REST API.

Before you begin

Before you start modeling the REST API, do a gap analysis on the existing Cúram functionality and identify any existing Cúram APIs that can be used to implement the functionality of the new REST APIs. As part of this gap analysis, identify the correct domain definitions to use for the resource representation.

What to do next

After completing the modelling of the Cúram rest façade and related structs, the java logic should be written to implement the APIs. The existing APIs identified during the gap analysis can be used to implement this logic.

Follow the normal Cúram build process to build the server and database and to refresh the Eclipse IDE environment.

Modeling the JSON representation

The JSON representation of a REST resource is modeled as one or more Cúram structs.

About this task

You must create a number of structs to represent the JSON structure for the following note resource:

```
{
  "note_id": "1234"
  "text": "The note text"
  "modificationHistory": [
      {"username":"sbrowne",
        "modificationDate":"2015-01-01"
      },
      {"username":"asmith",
        "modificationDate":"2015-05-01"
      }
  ]
}
```

Procedure

1. In the model, create the following new domain definitions, with the specified types.

Domain Definition Name	Base Cúram Type
NOTE_USERNAME	SVR_STRING
NOTE_DATE	SVR_DATE
NOTE_ID	SVR_INT64
NOTE_TEXT	SVR_STRING

2. Model a new struct, called ModificationHistory, to represent the modificationHistory property. This property contains nested content and each piece of nested content requires a new Cúram struct to be modeled. The ModificationHistory struct should contain the following parameters, with the relevant domain definitions, in the model:

Parameter Name	Domain Definition
username	NOTE_USERNAME
modificationDate	NOTE_DATE

3. Model a new struct, called Note, to represent the full note resource representation. This Note struct should contain the following parameters:

Parameter Name	Domain Definition
note_id	NOTE_ID
text	NOTE_TEXT

4. Create a 1-1 aggregation between the Note struct and the Modification struct, and set the aggregation role to be modificationHistory. The aggregation role is used in the JSON resource representation as the name of the property for the nested object.
5. Model a new struct, called NoteList, to represent a list of notes. This struct contains no parameters.
6. Create a one to many (1-*) aggregation between the NoteList struct and the Note struct and set the aggregation role to be data.
7. Model a new struct, called NoteIdentifer. This struct is not part of the note resource representation, but is required for the GET and DELETE methods to represent the note_id query parameter. The NoteIdentifier struct should contain the following parameter:

Parameter Name	Domain Definition
note_id	NOTE_ID

8. Model a new struct, called NotesQueryParameter. This struct is not part of the note resource representation, but is required for the GET method to represent the query parameters that can be passed to the method. The NotesQueryParameter struct should contain the following parameter:

Parameter Name	Domain Definition
username	NOTE_USERNAME

9. Complete the documentation field for all the structs and their parameters. This documentation is generated into a Swagger document that defines the REST API.

What to do next

Create a Cúram façade, with the stereotype rest, to implement the methods for the REST resource.

Related concepts

Cúram REST data types on page 29

A Cúram REST API resource request returns or accepts data in JSON format, which represents the various data types that are supported by Cúram.

Related reference

Cúram REST API configuration file on page 66

The REST configuration file is an XML file that defines the REST resources, including the versions and supported methods, in a REST API. One configuration file is defined per component and the file must be called `ResourcesConfig.xml`. It must be located in a `rest/config` directory within the EJBServer component.

Modeling the REST façade

REST API resource methods are implemented by Cúram modeled façade operations.

Before you begin

You must have created the required structs.

About this task

You must create a façade class, with a number of operations to represent the REST API resource methods defined in the following table.

Method	Resource	Comment
GET	/notes/	A collection resource that returns all the notes in the system
POST	/notes/	A collection resource to create a new note
PUT	/notes/{note_id}	A member resource to modify a specific note
GET	/notes/{note_id}	A member resource to return a specific note
DELETE	/notes/{note_id}	A member resource to delete a specific note

Procedure

1. In the model, create a façade called NoteAPI with a stereotype of rest.
2. For each entry in the following table, create a new operation on the NoteAPI façade, with the specified structs defined for input and output:

Façade Operation	Input Struct	Return Struct	Resource Method Mapping	Notes
readAllNotes	NotesQueryParameter	NoteList	/notes GET	The NotesQueryParameter struct defines the single username query parameter that is supported.
readNote	NoteIdentifier	Note	/notes/{note_id} GET	The NoteIdentifer input struct maps directly to the note_id path parameter.

Façade Operation	Input Struct	Return Struct	Resource Method Mapping	Notes
createNote	Note	NoteIdentifier	/notes POST	The NoteIdentifer output struct contains the note_id of the newly created note and is used to create the Location header returned.
modifyNote	Note	Note	/notes/{note_id} POST	The Note struct returned contains the update content.
deleteNote	NoteIdentifier	n/a	/notes/{note_id} PUT	The NoteIdentifer input struct maps directly to the note_id path parameter.

3. Complete the documentation field for the façade class and all the operations. This documentation field is used to generate a Swagger document that represents the definition of the REST APIs.

What to do next
Model the mandatory properties for each REST resource method.

Modeling the mandatory properties
Mandatory properties for a resource method are implemented in the Cúram model per façade operation.

About this task

The same struct is used for the resource representation across the GET, PUT, and POST methods, but can have different mandatory settings for the PUT and POST methods. Mandatory properties are specified in the model by using the mandatory fields setting on a façade operation. This allows us to meet the need to specify different mandatory properties for a PUT and POST method.

In the model, you set the Cúram mandatory fields property to reflect the list of mandatory properties. This property is a comma-delimited string, which lists out all the mandatory parameters of the input struct of a façade operation.

Mandatory fields that are needed for nested objects or lists can use the dot notation to indicate parent-child relationships. For example, to set the username property of a modification history entry to be mandatory, include modificationHistory.username in the mandatory fields option. This indicates that if an entry exists in the modificationHistory list, a username property must be specified or it results in an error.

You cannot set a mandatory option for the nested object or list parameter directly. For example, modificationHistory in the note example cannot be set as mandatory through the model.

To make a nested object or list field, such as an aggregation, mandatory, throw a Cúram AppException, with an error message that contains the name of the missing property, as part of the Java implementation of the façade operation.

Procedure

In the model, for each of the operations that are defined for the NoteAPI, set the mandatory fields value as outlined in the following table.

Façade Operation	Resource Method Mapping	Mandatory Fields Value	Notes
readAllNotes	/notes GET		
readNote	/notes/{note_id} GET	note_id	The note_id is needed to retrieve the member resource.
createNote	/notes POST	text	The text property is needed for creating a note. The note_id is automatically generated by Cúram.
modifyNote	/notes/{note_id} POST	note_id, text	Both note_id and text are needed to modify a note.
deleteNote	/notes/{note_id} PUT	note_id	The note_id is needed to identify the note to be deleted.

What to do next

Create the REST API configuration file, which maps the newly created façade operations to the associated REST API resource methods.

Cúram REST API modeling conventions

Cúram REST API resources are modeled by using Cúram facades with the rest stereotype. For each of the supported REST API resource methods, GET, POST, PUT, and DELETE, you must adhere to the following conventions.

Conventions for GET method facade

A rest façade operation that is mapped to a REST API resource's GET method should return a struct that represents the resource. The return struct is converted to a JSON object. The façade can have a maximum of one input struct. The input struct, if specified, should contain the path and query parameters that are supported for the REST API resource method. The attributes in the struct that represent the path parameters must match the name of the path parameters, as specified in the REST configuration file (`ResourceConfig.xml`), exactly. Any remaining attributes in the struct are considered to be supported query parameters and are also matched by name. Path parameters should be modeled as mandatory parameters by using the Cúram mandatory fields model option.

For example, the REST API GET method for the following URL has one query parameter, full_name, and requires the input struct for the corresponding façade to have a single full_name parameter.

```
https://host:port/Rest/v1/persons?full_name=Smith
```

The full_name specified as a query parameter, Smith, is mapped into the full_name parameter of the input struct and passed to the facade.

For example, a REST API GET method is defined in the *ResourcesConfig.xml* with the following path:

```
persons/{concern_role_id}/special_cautions/{special_caution_id}
```

{concern_role_id} and {special_caution_id} are the defined path parameters and the input struct for the corresponding façade must define two parameters of the same names. The value of concern_role_id and special_caution_id path parameters are mapped to the same named parameters of the input struct.

Conventions for a GET method that returns binary data

A REST API resource's GET method can return binary data in the response body. To request binary content, the Content-Type header of a GET request is set to anything other than application/json.

A façade operation that is used for such a REST API resource, must return a struct that contains the following named parameters with the specified domain definitions. All other attributes are ignored.

Parameter Name	Domain Definition	Required	Notes
data	SVR_BLOB	Yes	This parameter contains the binary content that is returned as the response body.
fileName	SVR_STRING	Yes	This parameter is used in the Content-Disposition header in the response. The extension of the file name is also used to determine the Content-Type header in the response.
contentType	SVR_STRING	No	Where specified, this parameter is used for the Content-Type header in the response. This replaces determining the Content-Type using the file name extension.

Parameter Name	Domain Definition	Required	Notes
metaData	SVR_STRING	No	Where specified, the parameter contains file metadata. The value that is specified is returned in the custom response header X-IBM-Curam-Metadata-Response. The value of the metaData that you assign in implementing the facade method must be a single JSON string that consists of name-value pairs. Examples are *`fileName`*, *`test.jpg`*, *`uploadedBy`*, and *`Test User`*. Before the REST infrastructure returns in the custom header, the REST infrastructure uses a URL to encode the values of the attributes because header values do not use utf8 encoding. Ensure that the consuming application of the REST API decodes the URL.

Conventions for POST method facade

A rest façade operation that is mapped to a REST API resource's POST method should have a single input struct that represents the resource. The request body content that is sent with a POST method is mapped into this input struct in the façade.

A POST method does not support query parameters.

Path parameters on a POST method must be part of the input struct that represents the resource. If the path parameters do not match the value of the same named property in request body, a 400 Bad Request HTTP response is returned.

The rest façade operation should return a struct that contains the unique ID for the newly created resource. This unique ID is appended to the POST request path to determine the value of the Location header property in the response.

Only the following domain definitions are supported for this single parameter in the return struct:

- SVR_INT64
- SVR_STRING
- SVR_UNBOUND_STRING

```
https://host:port/Rest/v1/notes
```

For example, a REST API POST method for the above URL, returns a Location header property in the following format:

```
https://host:port/Rest/v1/notes/{some_parameter}
```

Where some_parameter is the name of the attribute in the return struct from the corresponding façade operation.

Conventions for a POST method that must accept binary data

A REST API resource's POST method can accept binary data in the request body, instead of the default JSON content. To POST binary content, the Content-Type header of a POST request is set to anything other than application/json.

A façade operation that is used for such a REST API resource must define an input struct that contains the following named parameters with the specified domain definitions. All other attributes are ignored.

Parameter Name	Domain Definition	Required	Notes
data	SVR_BLOB	Yes	This parameter contains the binary content that is returned as the response body.
contentType	SVR_STRING	No	Where specified, this parameter is used for the Content-Type header in the response. This replaces determining the Content-Type using the file name extension.

Parameter Name	Domain Definition	Required	Notes
metaData	SVR_STRING	No	Where specified, the parameter contains file metadata. To call the API, the value must be sent in an X-IBM-Curam-Metadata custom request header. The value that you send in the custom header must be a single JSON string that consists of name-value pairs. Examples are `fileName`, `test.jpg`, `documentType`, and `Test User`. The following list outlines the requirements for calling the API: • A URL encodes the JSON string, before it is sent, to correctly encode any characters that are not in the ASCII format. • The REST infrastructure uses a URL to decode the values before the JSON string is assigned to the metaData attribute of the facade's input struct. The façade method that you implement must parse the JSON string and perform any validations on the details that it receives. The REST infrastructure removes any HTML script tags or other malicious content. The REST infrastructure also checks for well-formed JSON.

Conventions for PUT method facade

A rest façade operation that is mapped to a REST API resource's PUT method should have a single input struct that represents the resource. The request body content that is sent with a POST method is mapped into this input struct in the façade.

A PUT method does not support query parameters.

Path parameters on a PUT method must be part of the input struct that represents the resource. If the path parameters do not match the value of the same named property in request body, a 400 Bad Request HTTP response is returned.

The rest façade operation should return the struct that represents the resource. The return struct is converted to a JSON object.

Conventions for DELETE method facade

A rest façade operation that is mapped to a REST API resource's DELETE method must have a single input struct.

The input struct must contain only the parameters that represent the path parameters, which must match the name of the path parameters, as specified in the *`ResourceConfig.xml`* REST configuration file.

A DELETE method does not support query parameters.

Do not specify a return struct for a DELETE method.

Build validations

A set of build validations is available to help you to enforce adherence to standards and good coding principles. During development, you can optionally run the validations to ensure that the facades for REST APIs meet the required conventions and rules.

Errors are shown for facades that won't work with the REST engine.

Warnings are shown for issues that won't cause any error in usage, but that are considered best practice.

By default, the validations are turned off. You can enable the validations by specifying a command-line property at build time. For example:

```
build rest -Denable.validations=true
```

Related concepts

Cúram REST data types on page 29
A Cúram REST API resource request returns or accepts data in JSON format, which represents the various data types that are supported by Cúram.

Related reference

Cúram REST API methods on page 24
Cúram REST APIs support GET, POST, PUT, and DELETE methods on resources. The GET method is used to read data from Cúram. The POST method is used to create a resource. The PUT method is used to modify a resource, and the DELETE method is used to delete a resource.

Configuring the resource configuration files

You must create a REST configuration file in XML format to define the mapping of the REST resource paths to the Cúram façade operations.

Before you begin

Before the REST configuration file can be created, you must model and implement the Cúram façade class, which will have a stereotype of rest.

About this task

The REST configuration file defines all the resource paths for the API, and the methods that are available for each resource. Each resource method is mapped to a Cúram façade operation, and extra configuration values, such as mime-type and cache-control, can be set as required.

Creating a configuration file

Complete the following steps for creating the resource configuration XML file. The sample code that is defined for the Cúram notes REST APIs.

Procedure

1. From the *%CURAM_DIR%/EJBServer/components/<COMPONENT_NAME>/rest/config* directory, create a configuration file called *ResourcesConfig.xml*.

   %CURAM_DIR% is the Cúram installation directory, which by default is *C:\Merative\Curam\Development*.
2. Add the following content to the *ResourcesConfig.xml* file, which defines the GET method */notes* resource and includes the version number and a tag for the resource.

```
<?xml version="1.0" encoding="UTF-8"?>
<api>
  <version number="v1">
    <resource path="notes">
      <method verb="GET">
        <facade class="NoteAPI" method="readAllNotes"/>
        <tags>
          <tag>Note</tag>
        </tags>
      </method>
    </resource>
  </version>
</api>
```

3. Extend the */notes* `<resource>` element to include a second `<method>` to define the POST method. The child `<method>` elements define each of the methods that support the resource,

including the façade operation that provides the implementation. A `<resource>` element can contain multiple `<method>` elements.

```
<resource path="notes">
  <method verb="GET">
    <facade class="NoteAPI" method="readAllNotes"/>
    <tags>
      <tag>Note</tag>
    </tags>
  </method>
  <method verb="POST">
    <facade class="NoteAPI" method="createNote"/>
                <tags>
      <tag>Note</tag>
    </tags>
  </method>
</resource>
```

4. Add the path attribute for the resources. The path attribute for a `<resource>` defines the path that is used to access the resource and can include path parameters. Path parameters are denoted by curly brackets `{}`.

```
<resource path="notes/{note_id}">
  <method verb="GET">
    <facade class="NoteAPI" method="readNote"/>
    <tags>
      <tag>Note</tag>
    </tags>
  </method>
  <method verb="PUT">
    <facade class="NoteAPI" method="modifyNote"/>
                <tags>
      <tag>Note</tag>
    </tags>
  </method>
  <method verb="DELETE">
    <facade class="NoteAPI" method="deleteNote"/>
                <tags>
      <tag>Note</tag>
    </tags>
  </method>
</resource>
```

5. Optional: From the *`%CURAM_DIR%/EJBServer/components/<COMPONENT_NAME>/rest/config`* directory, create a properties file called *`RestConfig.properties`* and add a key value pair for a title property and a description property to the file.

   The title property defines the title of the REST API and the description property describes the purpose of the API. This information is included in the generated Swagger document, and defaults to the values:

```
title:Smarter Care & Social Programs REST API
description:This is the Smarter Care & Social Programs REST API.
```

6. Save the files.

What to do next

Build the REST APIs

Example: REST API configuration file

An example REST API configuration file (*ResourceConfig.xml*) for the Notes API resource.

```
<?xml version="1.0" encoding="UTF-8"?>
<api>
  <version number="v1">
    <resource path="notes">
      <method verb="GET">
        <facade class="NoteAPI" method="readAllNotes" />
                    <tags>
          <tag>Note</tag>
        </tags>
      </method>
      <method verb="POST">
        <facade class="NoteAPI" method="createNote" />
                    <tags>
          <tag>Note</tag>
        </tags>
      </method>
    </resource>
    <resource path="notes/{note_id}">
      <method verb="GET">
        <facade class="NoteAPI" method="readNote" />
                    <tags>
          <tag>Note</tag>
        </tags>
      </method>
      <method verb="PUT">
        <facade class="NoteAPI" method="modifyNote" />
                    <tags>
          <tag>Note</tag>
        </tags>
      </method>
      <method verb="DELETE">
        <facade class="NoteAPI" method="deleteNote" />
                    <tags>
          <tag>Note</tag>
        </tags>
      </method>
    </resource>
  </version>
</api>
```

Deploying Cúram REST APIs on Tomcat

After creating the REST configuration file, the next step in the process of creating a REST API is to build and run the API resources in the development environment by using Tomcat.

Before you begin

You must have completed the modeling, Java implementation, and creation of the REST configuration file for your REST resources.

Procedure

1. Set the CATALINA_HOME environment variable.
   This environment variable defines the home directory of the Tomcat installation, and is used to automatically deploy the REST API resources into Tomcat.
2. From the *%CURAM_DIR%/EJBServer* directory, run the following command-line command:

```
build rest
```

%CURAM_DIR% is the Cúram installation directory, which by default is *C:\Merative\Curam\Development*.

This target combines the defined REST resources from each component and deploys the REST API to the Tomcat web server in the development environment.

3. Using a web browser, open the following URL to confirm successful deployment of the REST API:

```
http://localhost:9080/Rest/api/definitions
```

Where 9080 is the default Tomcat port.

A list of the Swagger documents for each version of the API defined is displayed. Select a version to open the Swagger document for that version.

4. Using a web browser, open the following URL to confirm whether the /notes GET resouce method is working:

```
http://localhost:9080/Rest/v1/notes
```

A JSON object with an array of notes is displayed.

What to do next

Test the REST APIs using Junit or another unit testing tool.

Cúram REST API testing

Cúram REST resources are HTTP URL endpoints so you can take a number of different approaches to test your REST APIs.

Manual Testing

Cúram REST API resources can be manually tested directly in a browser, with browser plugins or with dedicated command line tools.

You can invoke PUT, POST and DELETE REST resource methods using a browser plugin. There are a number of REST plugins for common browsers like Chrome and Firefox. Using such a plugin, set the Referer and Content-Type headers as follows before making any requests:

- **Referer**
  curam://foundational.app
- **Content-Type**
  application/json

Automated Testing

It is good practice to develop automated unit tests for REST APIs and JUnit is one example tool that provides an approach for such testing.

In addition to JUnit, it may be worth considering the following open source libraries to make writing automated tests simpler:

- Jackson for JSON handling
- Apache HTTP Client.

In the absence of the above libraries, simple tests can be performed using the Java Standard Edition alone, for example:

```
URL url = new URL( "http://<host>:<port>/Rest/v1/<path>" );
HttpsURLConnection conn = ( HttpsURLConnection) url.openConnection();
// Verify the response code is what is expected
assertEquals(200, conn.getResponseCode());

StringBuilder responseBuilder = new StringBuilder();
BufferedReader in = new BufferedReader(new InputStreamReader(conn.getInputStream()));

while (in.ready()) {
        responseBuilder.append(in.readLine());
}
String expectedResponse = "some response";
assertEquals(expectedResponse, responseBuilder.toString());
```

Deploying a Cúram REST API on an application server

After testing the custom REST API resources, the final step is to deploy the Cúram REST API to an application server.

Before you begin

You must have completed the modeling, Java implementation, and creation of the REST configuration file for your REST resources.

You must have set up and configured a supported application server and database.

About this task

The *Rest.ear* file contains only the web application supporting the REST API. You must deploy the *Rest.ear* file to the application server where the Cúram server EAR file is deployed.

Procedure

1. From the *%CURAM_DIR%/EJBServer* directory, run the following command-line command to build the Cúram REST EAR file.

   ```
   build restEAR
   ```

   %CURAM_DIR% is the Cúram installation directory, which by default is *C:\Merative\Curam\Development*.
2. On IBM® WebSphere® Application Server, open the *%CURAM_DIR%/EJBServer/build/ear/WAS* directory and confirm that the *Rest.ear* file was successfully created.
3. Deploy the *Rest.ear* file to your application server. The *Rest.ear* file is platform neutral and can be deployed to WebSphere® Application Server, WebSphere® Application Server Liberty, or Oracle WebLogic Server. For example, deploy the *Rest.ear* file on WebSphere® Application Server by using the following command:

   ```
   build installapp -Dapplication.name=Rest -Dear.file=%CURAM_DIR%/EJBServer/build/ear/
   WAS/Rest.ear -Dserver.name=<server name>
   ```

Cúram REST API reference

These topics contain additional reference information to help you when working with Cúram REST API resources.

Cúram REST API configuration file

The REST configuration file is an XML file that defines the REST resources, including the versions and supported methods, in a REST API. One configuration file is defined per component and the file must be called *ResourcesConfig.xml*. It must be located in a *rest/config* directory within the EJBServer component.

All *ResourcesConfig.xml* files are combined with the facade and struct information from the Cúram model to generate a Swagger document. The Swagger document defines all the REST resources and methods supported for the API, and a Swagger document is generated for each version of the API.

The supported elements of the configuration file are outlined in the following sections.

Note: The order of the elements in *ResourcesConfig.xml* is defined in a corresponding XML schema file called *config-file-schema.xsd*.

The <api> element

The <api> element is the root element and groups all the resources for a REST API.

Required: Yes

Child elements: <version>

Attributes: None

The <version> element

The <version> element groups all resources for a particular version of the REST API.

Required: Yes

Child elements: <resource>

Attributes: See table.

Table 9: Attributes of the <version> element

Attribute	Description	Required	Default value
number	The version number of the REST API.	Yes	None

The <resource> element

The <resource> element defines a resource path, and groups operations on that resource.

Required: Yes

Child elements: `<method>`

Attributes: See table.

Table 10: Attributes of the `<resource>` element

Attribute	Description	Required	Default value
path	The name of the API path.	Yes	None

A path can contain path parameters, and these parameters are denoted by the use of curly brackets, {}. For example, the */notes/{note_id}* path contains one path parameter called note_id. When this resource is accessed, the {note_id} portion of the path is replaced with the actual note_id, for example, *notes/101*

The `<method>` element

The `<method>` element describes an operation on a resource.

Required: Yes

Child elements: <facade>, <consumes>, <produces>, <tags>, <cache-control>

Attributes: see table

Table 11: Attributes of the `<method>` element

Attributes	Description	Required	Default value	Values
verb	The HTTP verb for the operation.	Yes	None	• GET • POST • PUT • DELETE
response	Applies to POST requests only. Set to true to indicate that the response from the API contains a body, and a HTTP response code of 200.	No	false	true, false
no-content-response	Applies to POST requests only. Set to true to indicate that the response from the API contains neither a response body nor a location response header, and a HTTP response code of 204.	No	false	true, false

The <facade> element

The <facade> element describes the facade class and operation that implements the method. There is a one to one mapping between a facade method and an operation on a REST API resource method.

Child elements: <additional-error>

Required: Yes

Attributes: See table.

Table 12: Attributes of the <facade> element

Attributes	Description	Required	Default value
class	The corresponding facade class for the operation.	Yes	None
method	The corresponding operation within the facade class for the method.	Yes	None

The <additional-error> element

The <additional-error> element describes the specific HTTP error code that can be returned from a facade method, in addition to the regular 400 and 500 HTTP error codes. This element is optional and more than one <additional-error> can be defined where multiple error codes can be returned.

Child elements: None

Required: No

Attributes: See table

Table 13: Attributes of the <additional-error> element

Attributes	Description	Required	Default value
code	The HTTP error code that can be returned from the facade method.	Yes	None
description	A description of the error code.	Yes	None

The <produces> element

The <produces> element lists the mime-types that can be produced by a REST API resource method.

Required: No

Attributes: None

Child elements: <type>

The <consumes> element

The <consumes> element lists the mime-types that can be returned by a REST API resource method.

Required: No

Attributes: None

Child elements: <type>

The <type> element

The mime-type of the additional type that the resource method can use or produce.

Required: If the <produces> or <consumes> element is included, then at least one <type> element must be used.

Child elements: None

Attributes: None

The <tags> element

The <tag> element adds a key word tag to a REST API resource.

Required: No

Attributes: None

Child elements: <tag>

The <tag> element

The <tag> element groups related resources.

Required: Yes, if the parent <tags> element is included, at least one <tag> element must be used.

Child elements: None

Attributes: name, description

The <cache-control> element

The <cache-control> element contains the value for the cache-control header.

Specifying a <cache-control> element overrides the default settings for the entire cache-control header on a resource method. For example, the default specification of the codetables entity is:

```
<cache-control>private, max-age=604800, must-revalidate</cache-control>
```

However, if the configuration file includes the line,

```
<cache-control>max-age=3600</cachecontrol>
```

the cache-control header is cache-control: max-age=3600. Private and revalidate are no longer part of the header even though they were included in the default.

Child elements: None

Required: No

Attributes: None

The `<disable>` element

The `<disable>` element groups individual API endpoints that are to be disabled.

Child elements: `<path>`

Required: No

Attributes: None

The `<disable-recursive>` element

The `<disable-recursive>` element groups API namespaces or API versions that are to be disabled.

Child elements: `<path>`

Required: No

Attributes: None

The `<path>` element

The `<path>` element contains an API endpoint (including version), namespace, or version. If this element is used within a `<disable>` tag, then it must contain an API endpoint (including the version number), for example:

```
<path name="/v1/ua/persons" />
```

If this element is used within a `<disable-recursive>` tag, then it can contain either an API namespace (including the version number), or just the version number. For example:

```
<path name="/v1/ua" />
```

or

```
<path name="/v1">
```

Child elements: None

Required: No

Attributes: name

REST configuration properties

The Cúram REST infrastructure uses five properties. You enable the properties in the *Application.prx* file for your custom component or by using the Cúram administration console.

Table 14: REST configuration properties

Value	Description
curam.rest.refererDomains	This mandatory property configures a list of supported domains that you can set in the referrer header of a request. The property protects against Cross Site Request Forgery (CSRF) attacks. By default, the property is not set. In a deployed environment, the property must be set. Set the property as a comma-separated list of domains that are accepted in the referer header. For example, the value `abc.com, def.com` permits all requests with subdomains of `abc.com` and `def.com` that are set in the referrer header to successfully connect to Cúram REST APIs. When token-based CSRF protection is set, Cúram makes the policy more stringent and requires customers to also explicitly list host subdomains. **Note:** Any REST request that originates from a host domain or a subdomain that is not explicitly white listed in the referrer domains list is blocked. The property is not required for the Cúram mobile app or at development time. REST APIs accept a referrer header value that begins with *curam://* for mobile applications and accept the localhost domain at development time.

Value	Description
curam.rest.allowedOrigins	This string property contains a comma-separated list of allowed origins so that Cross-Origin Resource Sharing (CORS) requests from the origins are successful. `curam.rest.allowedOrigins` must contain the domain or partial domain of the app that is trying to access the REST APIs. For example, if the is deployed to https://example.abc.com:8080, then you must add *example.abc.com* or *abc.com* as a value for `curam.rest.allowedOrigins`. For development purposes, if the app is running on a `Node.js` server on *localhost*, then you must add *localhost* to `curam.rest.allowedOrigins`. For CORS requests, the REST toolkit automatically examines the value that is contained in the *origin* request header, which the browser sets automatically, and compares it to the values stored in `curam.rest.allowedOrigins`. If the values match, the necessary response headers that are needed to allow the CORS request to proceed are added to the response. If the values do not match, the browser automatically fails the CORS request because the required response headers are not included. The property is not required for mobile apps.
enable.rest.csrf.validation	This optional property enables token-based protection, which provides an enhanced level of security against cross-site request forgery (CSRF) attacks. By default, this property is disabled and the HTTP referrer header protects against CSRF attack vectors. The REST infrastructure supports token-based protection. Token-based protection provides an enhanced level of security against CSRF attacks. Use the `enable.rest.csrf.validation` property to enable the enhanced security level. By default, the property is disabled. Before you enable the property, ensure that you are familiar with CSRF. For more information about the HTTP referrer or CSRF, see the OWASP Cross-Site Request Forgery Prevention Cheat Sheet.
curam.rest.baseURI	The optional property configures an alternative base URI for the Location response header that is returned for POST request methods. If not enabled, the base URI from the request header is used. Use the property for web servers and gateways that change the context path of the REST API resources that are exposed to a client. Set the property to the full context path for the REST API, for example, someserver.abc.com:9123/Rest.

Value	Description
curam.rest.fail.on.unknown.field	This optional system property enables the generation of an error response when an unknown field is present in the body of a REST request. When enabled, the REST call returns a HTTP 400 response code with the following error message: 'The request contains an unrecognized attribute'. This property is disabled by default.

Related information

Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet

RestConfig.properties file

The *RestConfig.properties* properties file contains the properties for the title and the description of a REST API. This information is used for the title and description of the REST API in the generated Swagger document.

A developer can overwrite the default REST API properties and title by including a copy of the *RestConfig.properties* file in the *EJBServer/components/custom/rest/config* directory, and updating the values of the properties as required.

The default values for these properties are:

- *title=Smarter Care & Cúram REST API*
- *description=This is the Smarter Care & Cúram REST API*

Cúram REST API error handling

Refer to the following information to help you to design error handling for Cúram REST APIs.

HTTP error responses

Cúram REST API resource methods support a number of HTTP error responses, the most common of which are:

- HTTP 400 Bad Request status code

  This response code indicates an error that the user can recover from. For example, invalid data that is sent in the request body.
- HTTP 500 Internal Server Error status code

  This response code indicates an error that the user cannot recover from. In this instance, contacting the administrator is the only way to resolve the issue.

Cúram façade operations, which provide the implementation for a REST API resource method, can control the response code that is returned by throwing specific Cúram exceptions.

- curam.util.exception.AppException

  Throw an instance of the AppException, in the Java implementation, to cause an HTTP 400 status code response.
- curam.util.exception.AppRuntimeException

Throw an instance of the AppRuntimeException, in the Java implementation, to cause an HTTP 500 status code response.

In addition to the basic HTTP 400 and HTTP 500, it is possible to customize the AppException to cause the following HTTP error responses:

- HTTP 403 - Forbidden
- HTTP 404 - Not found
- HTTP 405 - Method not allowed
- HTTP 406 - Not acceptable
- HTTP 409 - Conflict
- HTTP 410 - Gone
- HTTP 412 - Precondition failed
- HTTP 416 - Requested Range Not Satisfiable
- HTTP 410 - Expectation Failed

To achieve this, the name of the catalog message used for the AppException should begin with "HTTP_4XX", where 4XX represents one of the supported 400 methods listed above. Any such exceptions will automatically result in the requested error code, as opposed to the default HTTP 400 for an AppException.

String identifier for application exceptions

The *AppException* exception type does not have a code identifier, instead it has a string identifier. To provide more than just an HTTP response code in the error response for this type of exception, the error response returned by the REST APIs also includes an additional attribute that contains the corresponding unique catalog string identifier of the underlying exception. This distinguishes between *AppExceptions* that result in the same HTTP response code.

Invalid path parameter example

If the path parameter value specified for an API resource GET method does not exist, the façade operation should throw an instance of curam.util.exception.AppException. The exception message should indicate to the user that the entered data was not found.

For example, the GET method for the /notes/{note_id} resource should throw an AppException with the following text if the note_id does not exist:

```
No record found for note_id %1s. Specify a valid note_id and try
 again.
```

If you want an HTTP 404 error to be returned in place of an HTTP 400 error, name the message catalog entry HTTP_404_NOTE_NOT_FOUND and use it to produce the AppException error. The result is an HTTP 404 error code response.

Configuration file changes

The HTTP 400 and HTTP 500 responses are included in the generated Swagger documentation by default. To ensure that additional HTTP response codes are included, for a particular REST API method, the REST configuration file can be updated to list the additional response codes supported.

For example:

```
<method verb="GET">

  <facade class="BasicPersonApi" method="readPerson">

    <addtional-error code="404" description="The requested data could not be found."/>

    <addtional-error code="403" description="Restricted access rights to the requested
 data."/>

  </facade>

</method>
```

Swagger and the Swagger UI

Swagger is an open specification for defining REST APIs.

The Swagger document specifies the list of resources that are available in the REST API and the operations that can be called on those resources. The Swagger document also specifies the list of parameters to an operation, including the name and type of the parameters, whether the parameters are required or optional, and information about acceptable values for those parameters. Additionally, the Swagger document can include JSON Schema that describes the structure of the request body that is sent to an operation in a REST API, and the .json schema describes the structure of any response bodies that are returned from an operation.

The Swagger UI is a tool that you can use from any web browser to visualize and test a REST API that is defined with Swagger. With the Swagger UI you can specify the inputs to an operation that is defined in that REST API, call that operation from the web browser, and inspect the results of calling that operation.

Related information

Swagger

Disable inbound REST APIs

Configure a list of APIs to be excluded by the REST infrastructure when it dynamically builds the REST APIs at run time.

Prerequisites

If you define custom REST APIs, then you already have a *`ResourcesConfig.xml`* file in a custom component. However, if you do not have custom REST APIs, create a *`ResourcesConfig.xml`* file in the following location: */EJBServer/components/<custom-component-name>/rest/config/ResourcesConfig.xml*

If you create a new file to satisfy the schema requirements, add an empty <api></api> element tag to the file.

Retaining *ResourcesConfig.xml*

Add the list of REST APIs to be disabled to *ResourcesConfig.xml*. *ResourcesConfig.xml* files can exist in any server component. However, to avoid overwriting *ResourcesConfig.xml* when you take on a new release, add the list to a *ResourcesConfig.xml* file in your custom component.

Methods to disable REST APIs

Disable REST APIs by using the following methods:

- Listing API endpoints individually
- Listing a namespace, all APIs defined with this namespace are disabled
- Listing a version, all APIs that are defined with this version are disabled

Disabling API endpoints individually

Identify if you need to disable any individual APIs, while leaving other APIs enabled within a namespace.

Add each API endpoint to a <disable> element in *ResourcesConfig.xml*. For example:

```
<disable>
 <path name="/v1/cwa/contact_logs" />
 <path name="/v1/cwa/contact_logs/{contact_log_id}/attachments" />
</disable>
```

Disabling APIs by namespace

Identify if you need to disable any APIs for an entire namespace.

Add each version and namespace to a <disable-recursive> element in *ResourcesConfig.xml*.

Note: You must add an entry for each API version in a namespace, you cannot disable all versions of a namespace by using a single entry.

For example, to disable all endpoints in the /cwa namespace, for version 1 and version 2 (assuming two versions of the APIs are defined out-of-the-box):

```
<disable-recursive>
 <path name="/v1/cwa" />
 <path name="/v2/cwa" />
<disable-recursive />
```

Disabling APIs by version number

Identify if you need to disable APIs for an entire version number.

Add each version path to a <disable-recursive> element. For example, to disable all version 1 APIs in the product:

```
<disable-recursive>
 <path name="/v1" />
<disable-recursive />
```

This example disables all default or custom APIs defined with v1.

Ordering elements in *ResourcesConfig.xml*

Add elements to *ResourcesConfig.xml* in the following sequence to match what is defined in the file's schema:

```
<disable>/<disable>
<disable-recursive></disable-recursive>
<api></api>
```

8.0.1.0 GraphQL

GraphQL is a query language for APIs.

Clients build queries for the data that clients need based on an underlying schema. The schema consists of a set of entities that are linked based on the business relationships between them so that the entities form a graph. APIs are also defined in the schema and represent an entry point to the graph.

The system means that a client can request only the data that the client wants. It also means that the client can combine what might be defined as multiple GraphQL APIs in the server into a single GraphQL query. As a result, significantly less network traffic passes between the client and the backend server.

In Cúram, a GraphQL API consists of a schema definition and an implementation class, called a data fetcher. The data fetcher wraps a facade operation from the server. A GraphQL query can be made over the Hypertext Transfer Protocol (HTTP), by using the GraphQL query language, to query one or more GraphQL APIs. The response is returned in a JSON format.

The GraphQL infrastructure is built into the REST application and is deployed as part of the REST ear.

GraphQL terms

Ensure that you understand the key terms that are associated with GraphQL.

The following table lists the common GraphQL terms that are used throughout the GraphQL documentation. Table 1. Common GraphQL terminology

GraphQL Term	Description
Query	The equivalent of a `read` or `GET` operation, where data is read and returned from the server.
Mutation	The equivalent of a `create`, `update`, `delete` or `POST`, `PUT` or `DELETE` operation, where data is modified on the server. Mutations are not supported by the GraphQL infrastructure.

GraphQL Term	Description
Schema	A schema is used to define data objects and the APIs that are the entry points to the objects. The schema permits validations of queries. The schema also permits your GraphQL APIs to be discovered and is analogous to a Swagger specification for REST APIs.
Data fetcher	The GraphQL schema is linked to an underlying Cúram facade operation by a Java™ class that is called a data fetcher. Data fetchers are also known as resolvers and third-party libraries. Data fetchers is a class that is started when a query runs and identifies how to retrieve or update data on the server, usually by calling to a facade operation in the EJB server.
Runtime wiring	The runtime wiring defines the data fetcher to use when an API that is defined in the schema is started at run time. You can also use the runtime wiring to specify different data fetchers for individual attributes of an object.
GraphQL endpoint	All queries and mutations use the same endpoint, that is, POST `https://<server>:<port>/Rest/graphql`. The details of the API or APIs to query and the attributes of the APIs to include in the response are all sent in the request body.

Note: By default, the GraphQL endpoint is disabled as GraphQL is a new feature and not yet widely used in Cúram. As a result, the GraphQL APIs in the product can't be accessed until the system property that controls the endpoint is enabled. For more information, see Configuring GraphQL properties.

Configuring GraphQL properties

To use GraphQL, you must first enable the GraphQL endpoint by using a system property. You can use other system properties to control features in the GraphQL server.

Enable the properties in the `Application.prx` file for your custom component or use the Cúram administration console. If you change these properties in a running server, you might need to restart the server to update the property cache in the REST application.

The following table provides information about the GraphQL configuration properties. Table 1. GraphQL configuration properties

GraphQL	Display name	Default value	Description
Enable GraphQL	curam.graphql.endpoint.enabled	disable	A setting that defines whether the GraphQL endpoint URL is enabled or disabled.

GraphQL	Display name	Default value	Description
Enable GraphQL introspection	curam.graphql.introspection.enabled	`True`	A setting that defines whether introspection queries are enabled or disabled. Introspection queries return details about the available GraphQL schema. The setting is required if you are using the GraphiQL HTML page to view the GraphQL schema or to test a GraphQL query for the APIs in a development environment. As the GraphiQL HTML page is not added to the REST ear, use the `False` default value where you are not supplying an integrated development environment (IDE) or other way to use introspection queries. In a production environment, set the property to `False`.
Maximum GraphQL schema query depth	curam.graphql.max.schema.query.depth	2	A setting that defines the maximum GraphQL schema query depth to prevent large queries that might potentially affect the performance of the server.
GraphQL schema complexity	curam.graphql.schema.complexity	20	Use the setting to define the complexity of a query because clients can query many APIs in one request. The setting defines the maximum complexity of a query that the server accepts.

Because the GraphQL server is a part of the REST application, the configuration properties for REST apply to GraphQL, in addition to the properties that are listed in Table 1. The configuration properties are listed at Cúram REST configuration properties

Procedure

The following steps outline how to set the configuration properties:

1. Log in to Cúram as a system administrator.
2. Click **System Configurations** > **Shortcuts** > **Application Data** > **Property Administration**.
3. From the **Name** field, enter `GraphQL` and click **Search**.
4. Click the **...** icon for the property.
5. Select **Edit Value...** to update the value.

6. Click **Save**.
7. Click **Publish**.

Developing a GraphQL API

Three artifacts are required to create a GraphQL API.

The following table lists the three artifacts that are involved when you create a GraphQL API.

Table 1. Artifacts that are involved when you create a GraphQL API

Artifact	Description
A GraphQL schema file entry.	The file entry defines the GraphQL API and the data objects that it returns.
Data fetcher Java™ class or Java™ classes.	The Java™ class or Java™ classes are the implementation classes that are started when an API is queried. The data fetcher calls to a facade method in the server and handles the mapping of input parameters to input structs.
A runtime wiring configuration file entry.	The file entry links the GraphQL API schema entry to a data fetcher class or classes.

Performance testing for new GraphQL APIs

GraphQL can be used to improve API performance through increased efficiency, by retrieving data in a single query. However, GraphQL is a flexible specification, with several well-documented caveats and anti-patterns. For this reason, it is advised that to prevent issues in production, you run performance tests on any GraphQL APIs that you create.

Modeling the GraphQL APIs

When you create a new GraphQL API, you must model a new facade in Rational Software Architect. You can also reuse an existing facade if a facade is available for the functionality that you want to expose by using GraphQL.

When you model the façade, ensure that the facade uses either the `<<rest>>` or `<<facade>>` stereotype when modeling the facade as both can be called from a data fetcher.

Note: When you use the `<<facade>>` stereotype, you cannot model lists within lists as it causes errors when you build the model.

Adding APIs to the GraphQL schema

You can create an entry in the GraphQL schema to define an API. To define APIs and responses in your GraphQL schema file, create a new schema file and then adding an API definition to a schema.

Creating a schema file

For each server component, only one schema file is added. If you do not have a schema file in your custom component, the following steps list outline how to create a schema file:

1. Create a `schema.graphqls` file in your custom component under the directory

`%CURAM_DIR%/EJBServer/components/<COMPONENT_NAME>/rest/graphql/config/schema.graphqls`

where:

- `%CURAM_DIR%` is the Cúram installation directory, which by default is `C:\Merative\Curam\Development`
- `<COMPONENT_NAME>` is the name of your custom component

2. Add a top-level entry that extends the `Query` type. The original `Query` type is already defined in a default schema. Duplicate entries are not permitted. However, the schema definition supports extensions of types.

Adding API definitions to a schema

The following steps outline how to add an entry for a new API into the schema:

1. Add an entry to the `Query` definition extension section, where the name of the API, the input parameters, and a response object type are defined.
2. Add a type definition for the response object of the API, where each attribute is given a name and type. The following list outlines important details about the definition and attributes:
   - As the response of the API corresponds to a return struct from a facade method, all attribute names must exactly match to the struct attribute names.
   - The type that is defined for each attribute is either a scalar or another object type.
   - Custom types are defined for special handling of certain data types.
   - For more information about the available data types, see Cúram data types and GraphQL scalars.
3. At build time, any schema files that exist in the EJBServer components are merged into a single schema file. Each schema file is validated for syntax and also to ensure the APIs and types are unique as no duplicate definitions are permitted.

Example

The following example represents one of the APIs in the schema. All types are prefixed by DOM to show that the types represent objects from the Domain model.

The `readIntegratedCase` API returns a `DOMIntegratedCase` object that contains a set of fields, where the `benefits` field is defined as an array of `DOMBenefit` objects.

In this way, elements in the schema are connected in a graph and the requester can explicitly specify the data that it wants to fetch. Each field in the data object type definition is either a scalar or another object. For more information about the scalars to use, see Cúram data types and GraphQL scalars.

Directives are used to denote special handling for code items. For more information about the details of directives, see GraphQL directives for code items and frequency patterns.

```
extend type Query {
    readIntegratedCase(case_id: GQL_ID!): DOMIntegratedCase
}

type DOMIntegratedCase {
        id: GQL_ID
        reference: String
        effective_date: GQL_Date
        registration_date: GQL_Date
        status: CodeItem @code
        type: CodeItem @code
        benefits: [DOMBenefit]
}

type DOMBenefit {
        id: GQL_ID
        reference: String
        type: CodeItem @code
        product_name: CodeItem @code
        product_type: CodeItem @code
        status: CodeItem @code
        effective_date: GQL_Date
}
```

Cúram data types and GraphQL scalars

An object type that is defined in a GraphQL schema consists of fields, where each field has a type that might be either a scalar or another object type itself. A scalar represents the lowest leaf of a query.

For more information, see Scalars and Schemas and Types.

List of available GraphQL data types and scalars

GraphQL provides a number of built-in scalars. The GraphQL server in Cúram provides some additional scalars to handle the Cúram specific Java types. When a query is invoked and data is read from the server, the following list outlines how the Java types are handled:

- The Java types are converted to their equivalent scalar.
- The scalars are serialized into JSON for the HTTP response.

It is important to define each field of an object in the schema with the correct scalar. The following table includes a list of the scalars to use for the corresponding Cúram and Java data types. The creation of new custom scalars is not supported.

Table 1. A list of the scalars to use for the corresponding Cúram and Java data types

Cúram data type	GraphQL scalars	JSON type	Description
SVR_STRING	String	String	A UTF-8 character sequence.
SVR_BOOLEAN	Boolean	Booelan	True or false.
SVR_INT8	Byte	Number	a java.lang.Byte based scalar.
SVR_INT16	Short	Number	a java.lang.Short based scalar.
SVR_INT32	Int	Number	A signed 32-bit integer.

Cúram data type	GraphQL scalars	JSON type	Description
SVR_INT64	GQL_ID	String	A Cúram scalar for converting a java.lang.long to a string because a JSON number cannot hold a value as large as a long value.
SVR_FLOAT	Float	Number	A signed double-precision floating-point value.
SVR_DOUBLE	Float	Number	A signed double-precision floating-point value.
SVR_DATE	GQL_Date	String, with the date in an ISO8601 format	A Cúram scalar for converting a Java `Date` to a Cúram `Date`.
SVR_DATETIME	GQL_DateTime	String, with the `datetime` in an ISO8601 format	A Cúram scalar for converting a Java `DateTime` to a Cúram `DateTime`.
SVR_MONEY	GQL_Money	Number	A Cúram scalar for Cúram Money to a double.

GraphQL directives for code items and frequency patterns

A directive in GraphQL indicates an extra configuration for a field.

The following list outlines the two Cúram directives in the GraphQL server:

1. The `@code` directive.
2. The `@frequency` directive.

These directives are used for the special handling of Java attributes that are used for code items and frequency patterns and that result in the Java attribute being converted into an object with multiple attributes.

The creation of extra directives is not supported.

Codes

A facade method might return a struct that contains an attribute that was modeled in RSA with a domain definition based on a CODETABLE_CODE definition to denote that it contains a value of a code item from a code table.

In the GraphQL schema, any fields in an object type definition that correspond to a struct attribute with a CODETABLE_CODE domain definition must be defined by using the `CodeItem` type and by using the `@code` directive. The following snippet outlines an example:

```
type Case
 id: GQL_ID
 status: CodeItem @code
```

As the `CodeItem` type is defined in the schema, it does not need to be added. The following snippet specifies the definition:

```
type CodeItem
  code: String
  description: String
  tablename: String
```

where `code` contains the code value, `description` contains a localized description of the code, and `tablename` contains the name of the code table to which the code belongs.

The following example shows how a JSON response for an API that includes how the preceding `Case` type might look:

```
"data": {
 "case": {
  "id": "123456789",
  "status": {
   "code": "RST1",
   "description": "Open",
   "tableName": "CaseStatus"
  }
 }
}
```

Frequency patterns

A facade method might return a struct that contains an attribute that was modeled in RSA with a domain definition based on a FREQUENCY_PATTERN definition to denote that it contains a value of a frequency pattern.

In the GraphQL schema, any fields in an object type definition that correspond to a struct attribute with a FREQUENCY_PATTERN domain definition must be defined by using the `FrequencyItem` type and by using the `@frequency` directive.

The following snippet outlines an example:

```
type Payment
 id: GQL_ID
 delivery_frequency: FrequencyItem @frequency
```

As the `FrequencyItem` type is already defined in the schema, it does not need to be added. The following snippet specifies the definition:

```
type FrequencyItem
  value: String
  description: String
```

where `value` contains the frequency pattern value and `description` contains a localized description of the pattern.

The following example shows how a JSON response for an API that includes the preceding `Payment` type might look:

```
"data": {
 "payment": {
  "id": "123456789",
  "delivery_frequency": {
   "value": "10010011",
   "description": "Recur every 234 week(s) on Monday"
  }
 }
}
```

Creating a data fetcher class

Each type in the schema is linked to a facade by a data fetcher Java class. The data fetcher class is a wrapper that performs any simple operations that are needed on the input parameters. The data fetcher then calls the facade.

One top-level data fetcher is mandatory for the query and is the implementation class that is started when the query is run.

You can also create extra separate data fetchers for the individual fields of a data object that is returned by an API. The method allows the data retrieval to be handled in the background by multiple facades. When a GraphQL API is queried, if the request does not include the field that uses a separate data fetcher, the corresponding facade is not invoked. As a result, less processing and fewer database reads might occur. However, if all fields are included in the request, the data fetchers are invoked synchronously. If the API is designed with too many separate data fetchers,invoking the data fetchers synchronously might negatively affect performance.

About this task

Each data fetcher class must implement a `graphql.schema.DataFetcher` interface. The `get()` method is the only one method that must be implemented. The following list outlines the tasks that the `get()` method must perform:

- Map the input parameters that are sent in the query to the input struct that the corresponding facade method must use.
- Call a utility method provided by the GraphQL server, passing it the name of the facade class and method that is to be called to retrieve the data and the populated input struct. In the background, this utility method calls the facade method in EJBServer. and retrieves the response struct from the facade method.
- Return a Plain Old Java Object (POJO) that matches what is defined as the query response object in the schema. Typically, the POJO is the response struct from the facade method if the name of the struct matches exactly to the name of the data object that is defined in the schema.

Procedure

The following pattern outlines how to create a data fetcher class file in your custom component:

`%CURAM_DIR%/EJBServer/components/<COMPONENT_NAME>/source/curam/<PACKAGE_NAME>/graphql/datafetcher/<CLASS_NAME>.java`

where:

- %CURAM_DIR% is the Cúram installation directory, which by default is `C:\Merative\Curam\Development`
- `<COMPONENT_NAME>` is the name of your component in the EJBServer
- `<PACKAGE_NAME>` is the name of your package in the EJBServer
- `<CLASS_NAME>` is the name of the Java™ class

Examples of data fetcher classes

Example 1: A single data fetcher for a GraphQL API

In this example, an API named `readIntegratedCase` is defined in a schema. The API returns a `DOMIntegratedCase` object, where one of the attributes is a further nested list of `DOMBenefit` objects.

The underlying facade method `DOMIntegratedCaseGQL.readIntegratedCase()` facade method in the EJBServer that the data fetcher class will wrap takes a `DOMCaseID` input struct. It also returns a `DOMIntegratedCase` struct, which contains a nested list of `DOMBenefit` structs. The facade method populates all the struct attributes, including the nested list of benefits.

This struct name and its attributes are a match for what is defined in the schema for the API.

The following code outlines the schema definition:

```
type Query {
    readIntegratedCase(case_id: GQL_ID!): DOMIntegratedCase
}

type DOMIntegratedCase {
        id: GQL_ID
        reference: String
        registration_date: GQL_Date
        benefits: [DOMBenefit]
}

type DOMBenefit {
        id: GQL_ID
        reference: String
        effective_date: GQL_Date
}
```

The following code sample shows how to implement a data fetcher for the `DOMIntegratedCase`:

```
public class IntegratedCaseDataFetcher implements DataFetcher<DOMIntegratedCase> {

  @Override
  public List<DOMBenefit> get(final DataFetchingEnvironment env) throws Exception {

    /* assign values to the input struct from the GraphQL request parameters, which are
 available from the DataFetchingEnvironment object.
    */
    final DOMCaseID domCaseID = new DOMCaseID();
    domCaseID.case_id = env.getArgument("case_id");

    final String facadeClassName = "DOMBIntegratedCaseGQL";
    final String facadeMethodName = "readIntegratedCase";

    final DOMBenefitList benefitList = (DOMIntegratedCase) GraphQLUtils
      .callServer(facadeClassName, facadeMethodName, inputStruct);

    return benefitList;
  }
}
```

The case ID is passed as a parameter by the query. The following code outlines how you can get the ID:

```
domCaseID.case_id = env.getArgument("case_id");
```

Example 2: Multiple data fetchers for a GraphQL API

Here, the same schema from the previous schema is used.

The following code outlines the schema definition:

```
type Query {
    readIntegratedCase(case_id: GQL_ID!): DOMIntegratedCase
}

type DOMIntegratedCase {
        id: GQL_ID
        reference: String
        registration_date: GQL_Date
        benefits: [DOMBenefit]
}

type DOMBenefit {
        id: GQL_ID
        reference: String
        effective_date: GQL_Date
}
```

However, for this example two underlying facade methods must be used to retrieve all the data for the API.

The `DOMIntegratedCaseGQL.readIntegratedCase()` method is still to be used to retrieve details of the integrated case, and it still returns a `DOMIntegratedCase` struct. However in this example the `DOMIntegratedCase` struct does not contain a nested list of `DOMBenefit` structs. The data fetcher implementation is identical to the first example.

Use a second facade method `DOMBenefitGQL.listBenefitsByIntCase()` to retrieve details about the benefits for an integrated case. It returns a `DOMBenefitList` struct, which contains a nested list of `DOMBenefit` structs. This facade method requires the benefit ID as an input.

The following code sample shows how a data fetcher for the list of benefits must be implemented:

```
public class BenefitListDataFetcher implements DataFetcher<List<DOMBenefit>> {

  @Override
  public List<DOMBenefit> get(final DataFetchingEnvironment env) throws Exception {

    /* assign values to the input struct from the GraphQL request parameters, which are
 available from the DataFetchingEnvironment object.
    */
    final DOMCaseID domCaseID = new DOMCaseID();
    final DOMIntegratedCase domIntegratedCase = (DOMIntegratedCase) env.getSource();
    domCaseID.case_id = domIntegratedCase.id;

    final String facadeClassName = "DOMBenefitGQL";
    final String facadeMethodName = "listBenefitsByIntCase";


    final DOMBenefitList benefitList = (DOMBenefitList) GraphQLUtils
      .callServer(facadeClassName, facadeMethodName, inputStruct);

    return benefitList;
  }
}
```

When the readIntegratedCase API is invoked, the benefit ID is not sent as a parameter in the request. The ID is contained in the `DOMIntegratedCase` struct that is returned by the higher-level data fetcher. The GraphQL server stores the responses of each data fetcher as it traverses through the nodes of the data objects in the `DataFetchingEnvironment` context object.

The following code outlines how the benefit ID can be retrieved from the response of the IntegratedCaseDataFetcher that is stored in the context object:

```
final DOMIntegratedCase domIntegratedCase = (DOMIntegratedCase) env.getSource();
domCaseID.case_id = domIntegratedCase.id;
```

Configuring the runtime wiring for APIs

Runtime wiring is required to link an API that is defined in the schema to an implementation in a data fetcher class. The details are defined in a runtime wiring configuration file.

About this task

The method indicates to the server the data fetcher to start at run time when a query is received, as defined in the runtime wiring configuration file.

Procedure

Creating a new runtime wiring configuration file

Only one runtime wiring configuration file is added per server component.

Where you do not have a runtime wiring configuration file in your custom component, the following steps outline how to create one:

1. Create a new `runtime_wiring.yaml` file in your custom component in the location `%CURAM_DIR%/EJBServer/components/<COMPONENT_NAME>/rest/graphql/config/runtime_wiring.yaml`

   where:

   - `%CURAM_DIR%` is the Social Program Management installation directory, which by default is `C:\Merative\Curam\Development`
   - `<COMPONENT_NAME>` is the name of your custom component.
2. Add a top level `Query` section if none exists.

Adding an entry to the runtime wiring configuration file

The following steps outline how to add an entry to the runtime wiring configuration file:

1. Add an entry to the `Query` section to define the data fetcher to be wired to the API. The following code provides an example:

   ```
   Query:
     - name: <the name of the API, as defined in the Query type in the schema>
       data_fetcher: <the fully qualified name of the data fetcher java class>
   ```
2. Optionally, further data fetchers can be wired to individual attributes within an API response object. Where none exists, add a top- level `FieldLevelWiring` section.

Add an entry to the `FieldLevelWiring` section to define the data fetcher to be wired to a particular attribute in an object type defined in the schema. The following code provides an example:

```
FieldLevelWiring:
  - object_type: <the name of object type, as defined in the schema>
    field:  <the name of the field within the object that the data fetcher applies
 to>
    data_fetcher: <the fully qualified name of the data fetcher java class>
```

3. At build time, the `runtime_wiring.yaml` files from all server components are merged into a single `runtime_wiring.yaml` file. Where duplicate entries are found for queries or fields, the entry in the component with the higher server component order takes precedence. The method permits a different data fetcher to be wired to an existing API, although the data fetcher must match the type definitions of the API in the schema.

Example

In example 2 in the Creating a data fetcher class page, the `readIntegratedCase` API is linked to the `IntegratedCaseDataFetcher`. A list of benefits is returned as part of the `DOMIntegratedCase` object and the `benefits` field is wired to its own data fetcher called `BenefitListDataFetcher`.

Generally, only fields that are linked to nested lists or objects have their own data fetchers. You can, however, also wire an individual field to its own data fetcher.

```
Query:
  - name: readIntegratedCase
    data_fetcher: curam.core.graphql.datafetcher.IntegratedCaseDataFetcher
FieldLevelWiring:
  - object_type: DOMIntegratedCase
    field:  benefits
    data_fetcher: curam.core.graphql.datafetcher.BenefitListDataFetcher
```

A further example of runtime wiring is in the `%CURAM_DIR%/EJBServer/components/core/rest/graphql/config/runtime_wiring.yaml` file.

Customizing data sources for existing GraphQL APIs

A number of existing GraphQL APIs are available to use. Customizing existing APIs and type definitions in the schema is not supported.

However, you can customize how the data sources for APIs are retrieved. A GraphQL API defined in the schema is linked to an underlying Cúram facade by using a Java™ class that is called a data fetcher. A runtime wiring configuration file contains the details of the data fetcher class or classes to use when a GraphQL API is called.

Before you begin

You must have access to a Cúram development environment. You must build the REST application to create the final merged version of the GraphQL schema file.

You must understand the essential elements of GraphQL APIs and artifacts, that is, schema file, data fetchers, and runtime wiring, including how to create them.

The following list outlines the elements that you must understand before you customize GraphQL APIs and artifacts:

- GraphQL
- Creating a data fetcher class
- Configuring the runtime wiring for APIs
- Modeling the GraphQL APIs
- Building the GraphQL APIs

Procedure

1. Identify the schema elements to customize. The following list outlines important considerations:
   - Identify the elements in the schema for which you intend to customize the data source. You can view the merged schema file directly in your development environment at `/EJBServer/build/RestProject/DevApp/WEB-INF/classes/curam/graphql/schema.graphqls`. However, it is easier to view the details of the schema using the GraphiQL in-browser html page that is included in the REST application on Tomcat. For more information, see Viewing the GraphQL queries by using the GraphiQL IDE.
   - Look in the runtime wiring file to identify the data fetchers that are currently wired to the schema element. The runtime wiring file is located in your development environment at `/EJBServer/build/RestProject/DevApp/WEB-INF/classes/curam/graphql/runtime_wiring.yaml`.
   - An entry in the `Query` section of the `runtime_wiring.yaml` exists to specify the top-level data fetcher for the API. There might also be entries in the `FieldLevelWiring` section where specific attributes of the schema elements are wired to their own data fetcher. It is important to identify all current data fetchers wired to the API and then identify the data fetchers that you want to change.
2. Create a custom Cúram facade. To customize the source of the data, you must create a custom Cúram facade in your own component. The new facade must match the same interface that the schema defines based on the existing facade.

   The input struct must contain attributes to match the request parameters. The output struct must be the same one as the one that is returned from the existing facade. The schema contains the name of the output struct that is being used because the name of the object type that is defined in the schema is also the name of the struct.

   The new custom facade can retrieve data from any part of the system if it complies with the same interface. Any `AppExceptions` that are produced by the facade are displayed directly in the application.
3. Create a custom data fetcher Java class to wrap the new Cúram facade so that it can be linked to from the schema. The data fetcher must parse the input from the GraphQL request and then map the input to structs so that the facade can be called. For more information, see Creating a data fetcher class.
4. Add an entry to the GraphQL runtime wiring configuration yaml file. The schema element is linked to the existing facade by the runtime wiring file. You must introduce your own version of this wiring file in your custom component. On a line-by-line basis, the wiring overrides the

matching wiring in the existing file. For more information, see Configuring the runtime wiring for APIs.

5. Build the REST application, which includes the GraphQL artifacts.

Example of customizing the data source for a GraphQL API

The following example shows how to customize the data source for an API and some API data object types that are defined in a schema.

The schema sample defines a `readIntegratedCase` API, that takes a `case_id` input parameter and returns an object of type `DOMIntegratedCase`. The sample shows that the `DOMIntegratedCase` type contains a set of fields where one field is linked to an array of `DOMBenefit` objects.

The following list outlines some field types:

- `GQL_ID` represents a unique identifier. `GQL_ID` is used for all IDs.
- `GQL_Date` represents a Cúram date type.
- `CodeItem` is used to define an object that represents a code item.
- `@code` is a directive, that indicates to the underlying server that a single Java attribute that contains a code value must be converted to a `CodeItem` object.

For information about the full list of scalars to use for each data type, see Cúram data types and GraphQL scalars.

The following code outlines the schema definition:

```
type Query {
    readIntegratedCase(case_id: GQL_ID!): DOMIntegratedCase
}

type DOMIntegratedCase {
        id: GQL_ID
        reference: String
        effective_date: GQL_Date
        registration_date: GQL_Date
        status: CodeItem @code
        type: CodeItem @code
        benefits: [DOMBenefit]
}

type DOMBenefit {
        id: GQL_ID
        reference: String
        type: CodeItem @code
        product_name: CodeItem @code
        product_type: CodeItem @code
        status: CodeItem @code
        effective_date: GQL_Date
}

type CodeItem {
        code: String
        description: String
        tableName: String
}
```

The following code shows the corresponding existing entry in the `runtime_wiring.yaml` file, that defines the two data fetcher classes to use for the `readIntegratedCase` API:

```
Query:
  - name: readIntegratedCase
    data_fetcher: curam.core.graphql.datafetcher.IntegratedCaseDataFetcher
FieldLevelWiring:
  - object_type: DOMIntegratedCase
    field:  benefits
    data_fetcher: curam.core.graphql.datafetcher.BenefitListDataFetcher
```

Customizing the data source

Two custom data fetcher classes are created, for example, `curam.custom.graphql.datafetcher.CustomIntegratedCaseDataFetcher` and `curam.custom.graphql.datafetcher.CustomBenefitListDataFetcher`. The classes reference custom facades. However, they must return the same object types as the existing data fetcher classes, that is `DOMIntegratedCase` and `List<DOMBenefit>`. The object types also match the object type definitions in the schema.

For more information, see Creating a data fetcher class.

In a `runtime_wiring.yaml` file that is located in an EJBServer custom component, an entry is created that references the two new custom data fetcher classes. The following code shows the entry in the `runtime_wiring.yaml` file:

```
Query:
  - name: readIntegratedCase
    data_fetcher: curam.custom.graphql.datafetcher.CustomIntegratedCaseDataFetcher
FieldLevelWiring:
  - object_type: DOMIntegratedCase
    field:  benefits
    data_fetcher: curam.custom.graphql.datafetcher.CustomBenefitListDataFetcher
```

For more information, see Configuring the runtime wiring for APIs.

At build time, because the custom component is listed higher in the `SERVER_COMPONENT_ORDER` variable, the entries that are in the custom component `runtime_wiring.yaml` file will overwrite the entries that are in the original component.

Building the GraphQL APIs

After you implement your GraphQL APIs, you must build and deploy the GraphQL APIs to a running application.

The following list provides a summary of the steps that are required to build and deploy the GraphQL APIs to a running application:

1. Build the model changes, if facades or structs are added.
2. Build and deploy the REST application. The REST application includes the GraphQL server and the GraphQL APIs. Different build targets exist for building and deploying to Tomcat and to an application server.
3. Enable the GraphQL endpoint by using the `curam.graphql.endpoint.enabled` system property. For more information, see Configuring GraphQL properties.

Building your model changes for a facade

When you make model changes, you need to run different build targets if you create a new facade to be used by your GraphQL API. If you reuse existing facades, you can skip this section.

About this task

Ensure that your local development environment is built as normal and your application can run successfully.

Modeled new facades and structs

The following list outlines the step to perform if you modeled a new facade class or operation in Rational Software Architect:

1. From `EJBServer`, run `build generated` to generate the code from the model. The step does not compile handcrafted code.
2. Create the implementation for your modeled facade operations.
3. Run `build compile.implemented`.
4. Run `build database` to insert the security identifiers that are associated with the facade operations into the database.

The following list outlines the steps to perform if you added or modified structs:

1. From `EJBServer`, run `build generated` to rebuild the server.
2. From `webclient`, run `build client` to rebuild the client, to regenerate the jars that contain the struct or structs.
3. From `EJBServer`, run `build rest` to rebuild the REST application and to copy the jars that contain the struct or structs.

Building and deploying GraphQL APIs on Tomcat

Use specific Ant targets to build and deploy the GraphQL server on Tomcat.

About this task

The GraphQL server is built into the REST application and includes the GraphQL APIs and other GraphQL artifacts. You must build the full REST application at least one time. You can use a separate GraphQL build target to update the GraphQL artifacts only, without the need to rebuild the full REST application.

Procedure

The following steps outline how to build the full REST application at development time:

1. Set your `$CATALINA_HOME` environment variable to the location of your Tomcat installation directory, for example `export CATALINA_HOME=%DEV_ENV_HOME%/tomcat`. **Note:** If you do not set your `$CATALINA_HOME` environment variable to the location of your Tomcat installation directory, the REST build target finishes successfully. However, a warning message is displayed in the console that indicates the variable was not set. As a result, nothing is built.
2. From `EJBServer`, run `build rest`. The step builds the REST development application in the `%CURAM_DIR%/EJBServer/build/RestProject/DevApp` directory. The following list outlines the step that are performed in addition to building the REST APIs:

- It merges any schema files from the server components that match the directory structure `/EJBServer/components/<COMPONENT_NAME>/rest/graphql/config/schema.graphqls`.
- It merges any runtime wiring files from the server components that match the directory structure `/EJBServer/components/<COMPONENT_NAME>/rest/graphql/config/runtime_wiring.yaml`. If duplicate entries are found in different files, the entries in the file with the higher server component order takes precedence.
- It compiles any data fetcher java classes from the server components are found in the directory pattern `%CURAM_DIR%/EJBServer/components/<COMPONENT_NAME>/source/curam/<PACKAGE_NAME>/graphql/datafetcher/<class_name>.java`.

Changing GraphQL artifacts

If you change any of the GraphQL artifacts only, that is, a data fetcher class, a schema entry, or a runtime wiring entry, then from `EJBServer`, run `build graphql`. Running that build target updates the GraphQL artifacts only in the built REST application.

Building and deploying GraphQL APIs on an application server

Use specific Ant targets to build and deploy your GraphQL APIs on an application server.

Procedure

The GraphQL artifacts are included in the REST ear for the deployment of your application on an application server.

To build the REST ear, run the build target `build restEAR`

For more information about deploying the ear to an application server, see *Deploying a Cúram REST API on an application server* and *Deploying Cúram*.

Viewing the GraphQL queries by using the GraphiQL IDE

GraphQL can process certain queries, called introspection queries, that return details about the schema.

You can use the GraphiQL IDE in a development environment, which sends an introspection query to the server and displays all the GraphQL APIs and their corresponding data objects. The application must be running in your development environment on Tomcat.

By default, introspection queries are disabled and must be enabled. For more information, see Configuring GraphQL properties.

The GraphiQL HTML page is not included in the deployed REST ear, so the schema is not exposed in production environments.

About this task

The GraphiQL IDE is available at `http://<server>:<port>/Rest/graphql.html`.

Viewing the queries structure

1. From the GraphiQL HTML page, the right side displays the Documentation Explorer for the GraphQL APIs that are available to query.

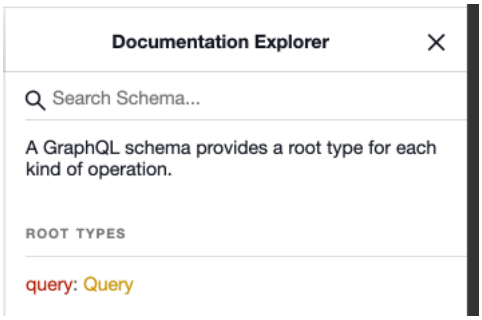


Figure 1. Documentation Explorer

2. Click on the `Query` root type to view the list of available fields. Click each of the APIs to view the details of the arguments that are being passed in and the type that is being returned.

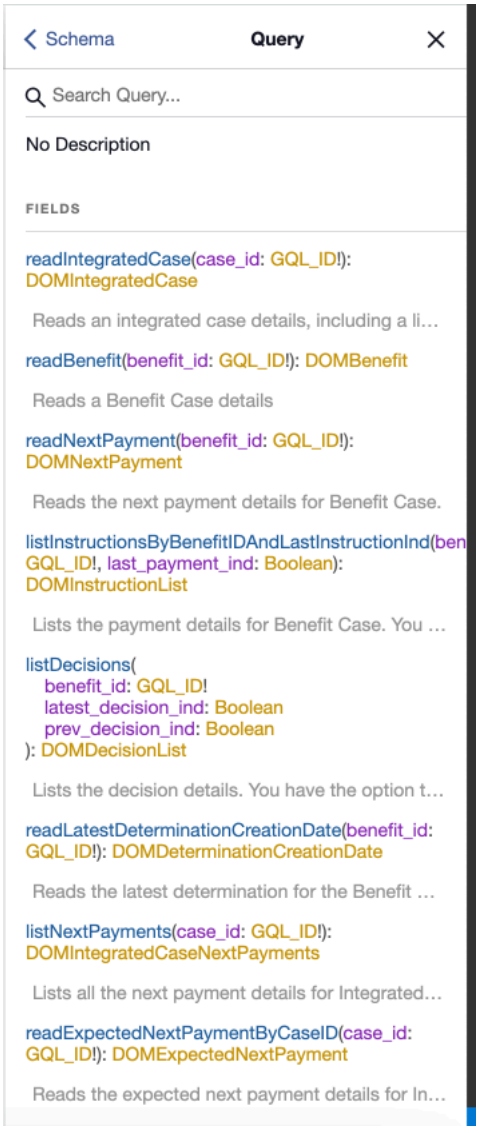


Figure 2. List of available fields

Testing a GraphQL query by using the GraphiQL IDE

You can use the GraphiQL IDE in a development environment to send a query to test the GraphQL APIs. The application must be running in your development environment on Tomcat.

By default, introspection queries, which are used by the GraphiQL IDE to retrieve the schema details from the GraphQL server, are disabled and must be enabled. For more information, see Configuring GraphQL properties.

The GraphiQL HTML page is not included in the deployed REST ear, so the schema is not exposed in production environments.

The GraphiQL IDE is available at `http://<server>:<port>/Rest/graphql.html`.

The following list outlines how to run a simple query and display the results:

1. From the GraphiQL left pane, start typing the API name that you want to call, for example `readIntegratedCase`. The tool automatically displays a drop-down list of the available APIs for you to choose.
2. Enter the arguments with the value. For example, specify the case identifier and the value for a case to query `case_id: "2012"`.
3. Enter the `{` open brace and click **Enter**.
4. Enter the attributes that you want to return for your query. The tool assists you with the list of attributes that are available for that API when you start typing.
5. Click the attributes that you want to use.
6. Click the  play button to run the query. The results from the query are displayed in the middle pane. The results are returned in JSON format.

Figure 1. The displayed query results

Sending a GraphQL query from a client

The server has only a single GraphQL endpoint, which accepts a POST request. The GraphQL server is included as part of the REST application. The URL of the GraphQL endpoint is `https://<server>:<port>/Rest/graphql`.

The details of the query, including the API or APIs to invoke and the attributes of the APIs that are requested, are added to the request body. The server then sends back a response that contains only the requested attributes.

The query is first parsed and validated by the GraphQL server to ensure all APIs and attributes that are specified match what is defined in the GraphQL schema, including any mandatory parameters, before any API is started.

Constructing a GraphQL query

A client decides what to add to a query by examining the GraphQL schema. The client then decides the APIs to call, and decides which attributes from the return object of the API to include in the response.

The details are sent in the request body when it calls the `POST https://<server>:<port>/Rest/graphql` endpoint.

Examples of a request body for a GraphQL query

The following basic schema example defines two APIs and the two data objects that the APIs return:

```
type Query {
 readIntegratedCase(case_id: GQL_ID): IntegratedCase
 readPerson(person_id: GQL_ID): Person
}

type IntegratedCase {
 id: GQL_ID
 reference: String
 registration_date: GQL_Date
}

type Person {
 id: GQL_ID
 name: String
 date_of_birth: GQL_Date
}
```

The following snippet outlines the code in the request body for a query that starts the `readIntegratedCase` API and that requests only the reference of the case:

```
{
 "query":"{
  readIntegratedCase(case_id: "2012"){
   reference
  }
 }
}
```

The following JSON outlines an example of a response body from the preceding query:

```
{
  "data": {
    "readIntegratedCase": {
      "reference": "203"
    }
  }
}
```

The following snippet outlines the code in the request body for a query that starts both the `readIntegratedCase` and `readPerson` APIs and that requests all attributes for both entities:

```
{
 "query":"{
  readIntegratedCase(case_id: "2012"){
   id
   reference
   registration_date

  }
  readPerson(person_id: "101"){
   id
   name
   date_of_birth
  }
 }
}
```

The following JSON outlines an example of a response body from the preceding query:

```
{
 "data": {
    "readExpectedNextPaymentByCaseID": null,
    "listNextPayments": null,
    "readIntegratedCase": {
      "id": "2012",
      "reference": "2012",
      "registration_date": "2020-04-14"
    },
    "readPerson": {
     "id": "101",
     "name": "Joe Bloggs",
     "date_of_birth": "1999-01-01"
    }
  }
}
```

For more information about how to create more complex queries and the use of fragments, see Queries and Mutations.

Security

When you access any GraphQL API in Cúram, you must be an authenticated user with valid authorization permissions for the relevant facade methods.

Authentication

Before you can make any GraphQL queries, you must authenticate with the Cúram application. For more information about authentication, see *Cúram REST API Security*.

Authorization

To access the API resource, authenticated users require sufficient authorization permissions. For users without permission, an error is returned in the query response body. Permissions are given to the underlying facades of a GraphQL API by using security identifiers (SIDs).

To better secure what authenticated users can access, remove SIDs from the database for any unused GraphQL API facade methods.

Cross Origin Request Forgery (CORS)

For security reasons, browsers restrict cross-origin HTTP requests that are initiated in a web application where the web application is hosted on a different server and port to the one where the GraphQL APIs are hosted.

However, the browser allows the HTTP request to proceed where, after it sends an initial OPTIONS request, the response from the server contains certain response headers that indicate to the browser that the request is allowed.

The REST infrastructure adds the response headers to an OPTIONS request from a browser where the domain that is set by the browser in the `origin` request header matches an allowlist of domains that are set in the `curam.rest.allowedOrigins` property.

For more information about the `curam.rest.allowedOrigins` property, see *Cúram REST configuration properties*.

Cross Site Request Forgery (CSRF)

Cross Site Request Forgery (CSRF) applies only where the GraphQL query comes from a web application. The first line of defense against CSRF is the referer header. The value for the referer header that is sent in the request is set by the browser. The value cannot be modified by JavaScript code. The value includes the domain of the server that the web application is hosted on.

A check is performed to ensure that the domain sent in the referer header matches an allowlist of domains that is set in the `curam.rest.refererDomains` property.

For any applications that are not web-based or for system-to-system communication, CSRF is not a factor. The referer header validation check passes where the value matches the scheme name of `curam://`. For example `curam://<name>`, where `<name>` can be any string name.

The REST infrastructure also optionally supports token-based protection as a second line of defense. For more information, see *Cross-Site Request Forgery (CSRF) protection for RESTful web services*.

Extra response headers

The following list outlines the response headers and values that are included in every response from a GraphQL request:

- X-XSS-PROTECTION=1; mode=block
- X-FRAME-OPTIONS=deny
- X-CONTENT-TYPE-OPTIONS=nosniff

The headers are used to combat cross-site scripting, click-jacking, and mime-type sniffing attacks.

Localization

GraphQL APIs can return certain data that is translatable into different languages. Examples include client error messages, code table descriptions, and other localizable text.

The language and locale is set for each GraphQL query that is based on the value that is sent in the `accept-language` request header. For example, if the `accept-language` request header is set to a value of `fr-CA` the server attempts to convert any localizable fields to the `fr-CA` locale. If no value is sent in the `accept-langauge` request header, then the server assumes a default of `en`.

HTTP supports multiple values to be set for the `accept-language` header. However, only the highest priority value is used and all other languages are ignored. When you use this header priority, set only one locale.

Error handling in GraphQL

Where an error occurs in GraphQL while a query is being processed, the response is still a `200 OK` response because many queries might be combined into one.

As a result, some queries might pass while some fail. Therefore, a single error response code cannot be used to reflect passing and failing. Instead, GraphQL adds an `errors` object, in addition to the `data` object, to the response body.

Example

In the example that follows, only one query was included in the request. Since no data was returned, the 'data' attribute is set to 'null'. If multiple queries were combined and only one failed, then the 'data' object contains the responses of the successful query while the 'errors' object contains the information of an unsuccessful query.

The underlying graphql libraries assign the values for the `message`, `locations`, and `path` attributes and the `classification` attribute inside the `extensions` object.

The `path` attribute contains the name of the query that was called and that is causing the error. The name of the query is useful where multiple queries were combined into one so that the query that caused the error can be identified.

The `message` attribute contains information that might be helpful for debugging. Do not display the information on a user interface.

The GraphQL server in the REST application adds information that is specific to the error to the `extensions` object.

```
{
  "errors": [
    {
      "message": "Variable 'case_id' has an invalid value. Unable to parse variable
 value as a Long",
      "locations": [
        {
          "line": 1,
          "column": 26
        }
      ],
      "path": "/readCase",
      "extensions": {
        "classification": "ValidationError"
      }
    }
  ],
  "data": {
    "readCase": null
  }
}
```

Structure of the error response

The error response differentiates between client and internal errors.

Client errors

Client errors contain localized messages to display on a user interface to the user. A list of error messages might be contained in the response.

The following list outlines one of the two places where the errors originate in the background:

- As an `AppException` that is produced from the underlying facade method in the EJBServer.
- As a `ConversionException` when it validates user input parameters against the domain definition for the corresponding input struct that is being sent to a facade method.

Errors that originate as `AppExceptions` contain a unique `string identifier`. Errors that originate as `ConversionExceptions` contain a unique `code identifier`.

The following code outlines the structure of the 'extensions' object:

```
"extensions": {
  "code":<top-level code of -130002 or -150601>
  "client_error": true
  "error_messages": [
    {
      "message": <localized message, suitable for displaying to end user>
        "code": <unique code, if originating as a ConversionException when validating
 user input params against domain definitions>,
        "message_id": <unique string id, if originating as an AppException>
      }
  ]
}
```

The following example outlines a full error response body for a client error:

```
{
  "errors": [
    {
      "message": "Exception while fetching data (/readIntegratedCase) : ERROR: The
 application server reported one or more exceptions",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "readIntegratedCase"
      ],
      "extensions": {
        "code": -130002,
        "client_error": true,
        "error_messages": [
          {
            "message": "You do not have maintenance rights for this case. Please
 contact your security administrator.",
            "message_id": "ERR_CASESECURITY_CHECK_RIGHTS"
          }
        ],
        "classification": "DataFetchingException"
      }
    }
  ],
  "data": {
    "readIntegratedCase": null
  }
}
```

For client errors, the cause of the error and the stack trace is logged only where tracing is set to a level of `trace_on` or higher in the `curam.trace` system property.

Internal errors

All other errors are considered internal errors. The cause of the error is not displayed to the user. Regardless of the cause of the error, the error code `-150600` is used.

To be consistent with the client errors, the structure of the extensions object still contains a nested list of messages. However, the list contains only one entry. The message is a localized generic error message to indicate that an error occurred. The top-level code of `-150600` is repeated for each error.

Regardless of the tracing level, the cause of the error and the stack trace is logged.

If tracing is set to a level of `trace_on` or higher, further information about the cause of the error is included for debugging purposes. The information varies depending on the exception type of the cause of the error.

The following code outlines the structure of the 'extensions' object:

```
"extensions": {
  "code": -150600
  "client_error": false
  "error_messages": [
    {
      "code": -150600,
      "message": <localized message to indicate something went wrong internally. May be
 displayed on a UI if desired.>
    }
  ]
}
```

The following code outlines an example of a full error response body for a client error:

```
{
  "errors": [
    {
      "message":"Cannot retrieve information at this time. Please contact your
 administrator.",
      "extensions":
        {
          "code":-150600,
          "client_error":false,
          "error_messages": [
            {
              "message":"Cannot retrieve information at this time. Please contact your
 administrator.",
              "code":-150600
            }
          ],
          "debug_info": {

 "root_cause_exception_type":"curam.rest.exception.CuramWebApplicationException",
            "root_cause_message":"The request is forbidden as the specified Referer
 header is not allowed.",
            "root_cause_code":-150210
          }
        }
      }
    ],
  "data": null
}
```

1.3 Developing outbound REST APIs

You can integrate Cúram with external applications that expose a REST API by making outbound API requests. Social Program Management includes the Jersey REST client library that you can use to make the outbound requests. Use this information to learn how to create and use the Jersey REST client.

Before you begin

The REST API that you are calling must use JSON format.

Getting started

Important considerations that you to need to know to get started. To create an outbound API and add it to Cúram, you must do the following tasks:

- Write a number of Java classes.
- If the API that you are calling requires authentication, you must add entries to the `Properties` database table to store credentials.
- Build the Cúram EAR and add third-party JARs that are supplied by Cúram to the application server and the class path of the Cúram application.

No further configuration is needed. The Ant targets to build the application server and deploy the Cúram EAR will automatically include your new Java classes.

Storing Java classes

When you develop an outbound API, you must add the Java files to the custom folder under the location that is represented by your SERVER_DIR environment variable, typically, `EJBServer/components/custom`. The code is packaged directly into the `Curam.ear`. Unlike the inbound REST APIs, outbound APIs are not packaged into the `Rest.ear`.

The following hierarchy shows where you can add your code for the outbound APIs. To keep the example simple, all classes are stored in the same package.

```
+ EJBServer
    + components
        + custom
            + <package-name>
               - <client-configuration> (required)
               - <outbound-api-classes> (required)
               - <custom-serializer-classes> (optional)
               - <custom-object-mapper> (optional)
               - <custom-module> (optional)
               - <client-request-filter> (optional)
```

Storing authentication credentials

If you need to authenticate with the API provider service, make sure you securely store your credentials. You must add new fields to the `Properties` database table and before you store any

passwords, ensure that you encrypt them. For more information about authenticating with an API provider, see Authenticating with the API service.

Building outbound APIs into the Cúram EAR

Use the server and EAR file build targets that are described in the *Cúram Server Developer* guide and the *Deploying Social Program Management* information for your platform.

The required third-party JARs are not automatically added to the Cúram EAR or class path. You must manually add them. For more information about the JARs that you must add to both a development environment and a deployed environment, see Building and deploying outbound APIs.

Serializing JSON and Java objects

To use the Jersey REST client, you must create plain old Java objects (POJOs) to match the JSON request and response body of the outbound API that you are calling. Each attribute in the Java object must exactly match the name of an attribute in the JSON object and have the appropriate corresponding type.

For more information about the API request and response bodies, see the documentation for the external API that you are calling.

The following example shows a simple JSON object and a corresponding Java POJO:

```
{
  "person":{
    "name" :"xyz"
    "age" : 20
  }
}
```

```
public class Person {

  private String name;
  private Integer age;

  //Getters and setters are also required.
}
```

You must also map nested JSON objects and arrays to Java complex objects and array lists. The Jersey REST client uses Jackson libraries to map the JSON request and response bodies of the API into Java objects. Jackson uses an `ObjectMapper` to convert between JSON and Java objects, where each attribute in a Java object is converted by a `serializer` or `deserializer` class.

Jackson provides default serializer and deserializer classes for all basic Java types. If you use the default serializer and deserializer classes, the following table shows which JSON type each Java type maps to and vice versa:

Java type	JSON data type
Long	Number
Integer	Number

Java type	JSON data type
Float	Number
Double	Number
String	String
Date	Timestamp
Datetime	Timestamp

If the JSON attributes in the API use the data types in the table, and you want to map them to the listed Java attribute types, you can allow Jackson to use its default processing. No further action is needed.

However, you might want to handle some types differently. For example, the API might define dates as JSON strings or you might want to map a JSON value to a Cúram Date type instead of a regular Java Date type. To do this, you must create custom serializer and deserializer classes.

Creating custom serialization

You can create custom Jackson serializer and deserializer classes to use instead of the default classes. Then, create a custom ObjectMapper to use your custom classes.

Creating custom serializer and deserializer classes

Create a custom class that extends a Jackson `StdSerializer` or `JsonDeserializer` class.

Note: When you register a serializer or deserializer class for a Java type, all attributes in a Java object of that type use the class when mapped to and from JSON. For example, if you create and register a custom serializer to handle Date values, all Java objects that contain an attribute of the type Date are serialized by using this class.

The following code shows how to write a serializer class that converts Java attributes with a Cúram Date type to a JSON string that contains an ISO8601 date:

```
public class CustomDateSerializer extends StdSerializer<Date> implements
 ContextualSerializer {

  /** Constructor. */
  public CustomDateSerializer() {
    super(Date.class);
  }

  /** Convert a Curam Date attribute to a string, and write to the JSON generator. */
  @Override
  public void serialize(Date date, JsonGenerator generator, SerializerProvider
 provider)
    throws IOException, JsonProcessingException {

    // handle null date value.
    if (date.equals(Date.kZeroDate)) {
      provider.defaultSerializeNull(generator);
      return;
    }

    Calendar calendar = date.getCalendar();
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    dateFormat.setTimeZone(TimeZone.getDefault());
    generator.writeString(dateFormat.format(calendar.getTime()));
  }

  @Override
  public JsonSerializer<?> createContextual(SerializerProvider arg0, BeanProperty arg1)
 throws JsonMappingException {
    return new CustomDateSerializer();
  }
}
```

The following code shows how to write a deserializer class that converts a JSON string that contains an ISO8601 date to a Java attribute with a Cúram Date type:

```
import curam.util.type.Date;

public class CustomDateDeserializer extends JsonDeserializer<Date> implements
 ContextualDeserializer {

  /** Deserialize the content. */
  @Override
  public Date deserialize(JsonParser parser, DeserializationContext context)
      throws IOException, JsonProcessingException {

    final String dateString = parser.getText();

    // If the JSON string value is empty, this should be mapped to a Curam empty date.
    if (StringUtil.isNullOrEmpty(dateString)) {
      return Date.kZeroDate;
    }

    try {
      // ISO8601 format is 'yyyy-MM-dd'.
      final SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

      // construct a Curam date.
      dateFormat.setTimeZone(TimeZone.getDefault());
      return new Date(dateFormat.parse(dateString).getTime());

    } catch (final ParseException e) {
      // return a valid default value to continue processing.
      return Date.kZeroDate;
    }

  }

  /** Handle null values. */
  @Override
  public curam.util.type.Date getNullValue() {
    return curam.util.type.Date.kZeroDate;
  }

  /** Simply need to return a new instance of this class. */
  @Override
  public JsonDeserializer<?> createContextual(DeserializationContext
 deserializationcontext, BeanProperty beanproperty)
    throws JsonMappingException {

    return new CustomDateDeserializer();
  }
}
```

Registering the custom classes with Jackson

After you write your serializer and deserializer classes for any types that you want to handle differently to the default Jackson classes, you must register them with Jackson. Do this by creating a Jackson `SimpleModule` class and a Jackson `ObjectMapperProvider` class.

The following code shows how to create a custom SimpleModule class that adds the custom date serializer and deserializer classes that are shown in the previous examples:

```
public class CustomSimpleModule extends import
 com.fasterxml.jackson.databind.module.SimpleModule {

  public CustomSimpleModule() {

    addSerializer(curam.util.type.Date.class, new CustomDateSerializer());
    addDeserializer(curam.util.type.Date.class, new CustomDateDeserializer());
  }
}
```

Note: You register the serializer and deserializer classes for a Java type so that every Java attribute of that type is mapped to and from JSON by using the same serializer and deserializer classes. You cannot register multiple serializer or deserializer classes for the same Java type.

The following code shows how to create an `ObjectMapper` class and how to register the `CustomSimpleModule` class.

```
import javax.ws.rs.ext.ContextResolver;
import com.fasterxml.jackson.databind.ObjectMapper;

public class CustomObjectMapperProvider implements ContextResolver<ObjectMapper> {

    public CustomObjectMapperProvider() {

        customObjectMapper = new ObjectMapper();
        customObjectMapper.registerModule(new CustomSimpleModule());

        // can also choose to enable or disable various features:
        defaultObjectMapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);

    }
}
```

In addition to registering the `CustomSimpleModule` class, you can set or disable various Jackson features in the ObjectMapperProvider class. The previous code sample also shows how to disable the Jackson feature that writes dates as datestamps.

For more information about the Jackson features that you can enable or make unavailable, see SerializationFeature and DeserializationFeature in the Jackson documentation.

Creating the Jersey REST client

Create a Jersey REST client to use to make an outbound request by configuring various settings. Then, use the configuration to create a client instance.

Configuring the REST client

Configure a Jersey REST client to:

- Register any custom providers that you created.
- Set any properties as needed.

The following code shows how to create the configuration and how to register the `CustomObjectMapperProvider` class that was created in the example in the previous section. Register custom providers only if you are using custom serializer and deserializer classes instead of the default Jackson ones.

```
ClientConfig clientConfig = new ClientConfig();
clientConfig.register(new CustomOutboundObjectMapperProvider());
```

You can set extra properties if needed, for example, a read timeout and a connection timeout for slow connections. The following code shows how to set these properties:

```
clientConfig.property(ClientProperties.READ_TIMEOUT, readTimeoutValue);
clientConfig.property(ClientProperties.CONNECT_TIMEOUT, connectionTimeoutValue);
```

For more information about the client properties that you can set, see Class ClientProperties in the Jersey Javadoc documentation.

If you want to use the same properties and `ObjectMapper` class for all outbound API requests, therefore the same serializer and deserializer classes, only one instance of a ClientConfig class is required.

Note: By default, HttpUrlConnection provides the transport layer in the Jersey REST client. You can configure the Jersey client to use an alternative transport connector, for example, the Apache HTTP client through a ApacheConnectorProvider class. Alternative connector implementations require additional Jersey libraries that are not included in Cúram and are outside the scope of this information. For more information about alternative connector implementations, see *Client Transport Connectors* in the Jersey documentation.

Creating the REST client

Creating and disposing the Jersey REST client is a heavyweight process. Therefore, it is important to use a limited number of client instances. You do not need to create a new client instance for each outbound API request. You can reuse the same client instance for multiple outbound requests.

The following code shows how to create an instance of the REST client by using the ClientConfig class that was created in the previous example.

```
Client client = ClientBuilder.newClient(clientConfig);
```

Optionally, you can also add a request filter to a client. For more information, see Adding client request filters

Full example of how to configure and create a REST client

The following code shows the previous code samples combined into a full example that includes the following methods:

1. A method to create a client configuration, with the following details:
   - Registers a custom ObjectMapperProvider. The assumption is that this class is already created and custom serializer and deserializer classes are registered for certain Java types.
   - Sets read timeout and connection timeout property values.
2. A method to create a REST client by using the client configuration.

```
public ClientConfig createClientConfig() {

    org.glassfish.jersey.client.ClientConfig clientConfig = new ClientConfig();

    clientConfig.register(new CustomOutboundObjectMapperProvider());
    clientConfig.property(ClientProperties.READ_TIMEOUT, readTimeout);
    clientConfig.property(ClientProperties.CONNECT_TIMEOUT, connectionTimeout);

    return clientConfig;
}

public Client createClient() {

 javax.ws.rs.client.Client client = ClientBuilder.newClient(createClientConfig());
    return client;
}
```

Making an outbound API request that uses the REST client

After you configure and create the Jersey REST client, use it to send an outbound request and to map the response, if any, to a Java object.

Constructing an outbound request

First, construct the request by adding the details of the URL that you want to call to a `WebTarget` instance. You can add the following details:

- The URL of the request.
- Any path parameters.
- Any query parameters (for GET requests only).

The following code shows how to construct a WebTarget instance and how to add path parameter and query parameters:

```
WebTarget webTarget = client.target("https://example.com")
       .path("/api")
       .path("/resource")
       .queryParam("include_inactive", "true");
```

Add the remaining details of the request to an `InvocationBuilder` instance. You can add the following details:

- Any headers and their values to include in the request.
- The expected media type of the body of the response.

The following code shows how to construct an InvocationBuilder instance, how to add an `Accept-Language` header, and how to specify that the response body of the API is `application/json`:

```
Invocation.Builder invocationBuilder = webTarget
          .header("Accept-Language", "en-US")
          .request(MediaType.APPLICATION_JSON);
```

Sending an outbound request

Call the API by specifying the HTTP method for the request on the `InvocationBuilder` instance. The following example shows how to make the GET request and the response that is returned:

```
Response responseFromAPI = invocationBuilder.get();
```

For a POST or PUT request that contains a request body, you also pass in the Java object that is mapped to the JSON request body. The following example shows a POST request that passes in a `myObject` instantiated POJO, and specifies that the request body media type is `application/json`:

```
Response responseFromAPI = invocationBuilder.post(Entity.entity(myObject,
 MediaType.APPLICATION_JSON));
```

Handling the response from the request

Check the HTTP response code that determines whether the request was successful or not. For a successful response, if the response contains a response body, it is deserialized to a Java object. For the HTTP code that denotes a successful response, see the documentation for your API. For example, 200 is the standard code for a GET request. A 200 or 201 code might be returned from a POST request. A 204 code might be returned for a POST or DELETE request and means that no response body exists.

If the request is not successful, the response body usually contains an error response. Therefore, make sure you check the response code first for a success code before you attempt to deserialize the body. The deserialization of the response body fails if it tries to deserialize into the wrong POJO type.

In many cases, if the request is not successful, the HTTP error response code provides sufficient information about the problem. However, you can also deserialize the error response body from the API into a Java object if you need further details. You must define a Java object to match the JSON format of the error response. For information about the format, see the documentation for the API that you are calling.

Important: If you call the `Response#readEntity()` method, the connection is automatically closed. However, you must manually close the connection if you don't call the `Response#readEntity()` method. For example, you might not call the `Response#readEntity()` method if the response has no response body or if the API returns an error response body that you do not read.

The following code shows how to check the HTTP response code before you deserialize the response body. It assumes that the API returns a code of 200 if successful, and that the response body must be deserialized into a Java object of type `MyResponseObject`.

```
if (HttpStatus.OK.value() == responseStatus) {

 //response body
    MyResponseObject responseObj = responseFromAPI.readEntity(MyResponseObject.class);

    //response headers (if required)
 MultivaluedMap<String, Object> headers = responseFromAPI.getHeaders();

} else {
    // handle the error
    // if not reading the error response entity, you MUST close the connection:
    responseFromAPI.close();
}
```

In an error scenario, you might want to log or handle different errors in different ways. For example, a 401 response code means unauthorized, so you might need to log in again or refresh your authentication token. A 301 response code means a redirect, which you might want to follow if the API service is trusted. See your API documentation for the different error response codes that the API returns.

Note: If you do not know which attributes a response body contains and you cannot create a Java POJO to match, you can use a generic Java object. Jackson maps the JSON object to a Java Map of name-value pairs.

```
Object genericObject = responseFromAPI.readEntity(Object.class);
```

Full example of how to make an outbound API request

The following example shows the previous code examples that are combined to make a GET request and to handle the response:

```
public ResponseObject invokeOutboundAPI() {

 javax.ws.rs.client.WebTarget webTarget = client.target("https://example.com")
       .path("/api")
       .path("/resource")
       .queryParam("include_inactive", "true");

 javax.ws.rs.client.Invocation.Builder invocationBuilder = webTarget
          .header("Authentication", "Bearer " + accessTokenValue)
          .request(MediaType.APPLICATION_JSON);

    javax.ws.rs.core.Response responseFromAPI = invocationBuilder.get();

    if (200 == responseStatus) {

  //response body
     MyResponseObject responseObj = responseFromAPI.readEntity(MyResponseObject.class);

 } else if (401 = responseStatus) {

  responseFromAPI.close();
  //perform logic to reauthenticate or to use refresh token to get new access token

 } else {

     // if not reading the error response entity, you MUST close the connection:
     responseFromAPI.close();

     throw new RuntimeException("outbound api request failed");
 }
}
```

Authenticating with the API service

To access an API, you might need to first authenticate with the API service. Typically, the documentation for the API that you are calling details how to authenticate. Authentication is normally a two-step process. First, you call a specific authentication URL and receive back a token. Then, you send the token with each outbound API request. When the token expires, you must obtain a new token.

Authenticating and retrieving an access token

You can retrieve the authentication token programmatically by making a POST request to a specified URL, passing in credentials, and receiving back an access token. You can make the POST request by using the REST client in the same manner as any other outbound API request.

Note: You must store the credentials that you use to authenticate in a secure manner. For example, if you use a system username and password to authenticate, encrypt your password and store it on the Properties database table. Then, decrypt the password before you send the authentication request.

A response from an authentication URL might be similar to the following example:

```
{
  "access_token": "BsT23OjbzRn430xzMLgV3Ia",
  "token_type": "bearer"
}
```

When you retrieve the access token, make sure you temporarily retain it because you must add it to a request header and send it with each outbound API request.

Sending an access token in the outbound API request

The most common request header that is used to send an access token is the `Authorization` header.

The following code shows how to add a bearer token (received from the authentication request) to an outbound API request, by using the `Authentication` header with the `Bearer` prefix. You add the bearer token to the `InvocationBuilder` instance.

```
Invocation.Builder invocationBuilder = webTarget
          .header("Authentication", "Bearer BsT23OjbzRn430xzMLgV3Ia" )
          .request(MediaType.APPLICATION_JSON);
```

Refresh tokens

Some authentication services also include a refresh token and expiry time with the access token. The response from the authentication URL that includes a refresh token might be similar to the following example:

```
{
  "access_token": "BsT23OjbzRn430xzMLgV3Ia",
  "refresh_token": "RY2sT23OjR30xzMLgV3u7c",
  "token_type": "bearer",
  "expires": 3600
}
```

Make sure you temporarily retain the refresh token along with the access token. When the access token expires, you can make a POST request to the authentication URL that includes the refresh token. A new valid access token is then returned.

The simplest way to handle the flow of an expired token with a refresh is to programmatically check the response from the API. A 401 response code is normally returned if the token expires. The response likely includes an `invalid_token` error code. You can send the refresh token to the authentication URL, which returns a new valid access token. Then, you can make another request to the API with the new access token.

The refresh token has an expiry time. You can reuse it to get new access tokens until it is no longer valid. Then, you must call the authentication URL with the credentials that are passing in. The API response usually contains a code or other differentiator so that you can programmatically determine when you need to send the refresh token, and when you need to resend credentials.

Building and deploying outbound APIs

Use Ant targets to build the outbound APIs into the Cúram application EAR file and deploy them.

The required Jersey JARs and their dependency JARs are provided in Cúram. However, they are included only in the `Rest.ear` file and the REST development application. Outbound APIs are built and deployed into the Cúram EAR file. Therefore, you must add the required JARs to the following locations:

- Your EJBServer project class path in a development environment.
- Your `EJBServer/components/custom/lib` directory, for inclusion in Ant scripts and the Cúram EAR file.

The following list outlines the required JAR files that you must add and their locations:

- `/EJBServer/components/Rest/restlib/dependencyLibsCore/hk2-api-<version>.jar`
- `/EJBServer/components/Rest/restlib/dependencyLibsCore/hk2-locator-<version>.jar`
- `/EJBServer/components/Rest/restlib/dependencyLibsCore/hk2-utils-<version>.jar`
- `/EJBServer/components/Rest/restlib/dependencyLibsCore/jersey-client-<version>.jar`
- `/EJBServer/components/Rest/restlib/dependencyLibsCore/jersey-common-<version>.jar`
- `/EJBServer/components/Rest/restlib/dependencyLibsCore/jersey-guava-<version>.jar`
- `/EJBServer/components/Rest/restlib/dependencyLibsCore/javassist-<version>.jar`
- `/EJBServer/components/Rest/restlib/dependencyLibsCore/jackson-jaxrs-base-<version>.jar`
- `/EJBServer/components/Rest/restlib/dependencyLibsCore/jackson-jaxrs-json-provider-<version>.jar`
- `/EJBServer/components/Rest/restlib/dependencyLibsCore/jackson-module-jaxb-annotations-<version>.jar`

Building the outbound REST APIs in your development environment

1. Set a J2EE_JAR Eclipse class path variable. For more information about how to set this variable, see the *Installing a Development Environment* guide.
2. Add the required Jersey JAR files to your class path for the EJBServer project. In Eclipse, right-click the EJBServer project in the project explorer view, then click *Build Path -> Configure Build Path -> Add External Jars*.

Building the application by using Ant targets

1. Set a J2EE_JAR environment variable that points to the installed Java EE JAR file. For more information about how to set this variable, see the *Installing a Development Environment* guide.
2. Copy all required Jersey JAR files to your `EJBServer/components/custom/lib` directory. Placing them in this directory ensures that the JAR files are automatically added to the Java compile classpath and the Cúram EAR file at build time.
3. Use the standard `server` Ant target to compile and include the code for the outbound APIs in the application. To package the compiled outbound APIs into the `Curam.ear` with all other server code, use the standard `websphereEAR` or `weblogicEAR` Ant targets.

Deploying to an application server

After you build your Cúram EAR file, you can deploy the application in the usual way.

If a LinkageError or other class loading conflict error occurs at runtime that indicates the `javax.ws.rs.core.Response.class` is already loaded, delete the `jsr311-api-1.0.jar` from the Cúram EAR file.

The JAX-RS 1.0 specification APIs and interfaces are contained in the `jsr311-api-1.0.jar` and these might conflict with the JAX-RS specification APIs and interfaces that are supplied by the application server. Apache Axis2 provides the `jsr311-api-1.0.jar`, but is not used by the Cúram application or by SOAP web services. Therefore, you can remove it.

Configuring a WebSphere application server to use Jersey

WebSphere Application Server version 9 and higher provides an implementation for JAX-RS that is based on Apache CXF. By default, Websphere is configured to use the Apache CXF implementation instead of any other JAX-RS implementation that is provided in the application.

You must configure WebSphere JAX-RS provider to provide support for the 2.0 specification only. This allows WebSphere to use the Jersey implementation that is packaged in the Cúram EAR instead of Apache CXF.

Ensure that you set the JAX-RS provider to **2.0 spec**. For more information, see Coexistence of JAX-RS 2.0 with JAX-RS 1.1.

Troubleshooting when adding JARs to your application server or Tomcat environment

A list of issues that you might experience when you add the JARs to an application server and how to resolve them.

- The list of required JARs does not contain the `javax.ws.rs-api-2.0.jar`. The `javax.ws.rs-api-2.0.jar` contains the interfaces for the JAX-RS 2.0 specification that Jersey implements. Although these interfaces are required, most versions of the application server already contain the `javax.ws.rs-api-2.0.jar` or a similar JAR that includes the JAX-RS 2.0 specification classes. If you experience a `ClassNotFoundException` for any classes in a `javax.ws.rs.*` package, check whether your application server includes the `javax.ws.rs-api-2.0.jar`. If not, add it with the other required JARs. The `javax.ws.rs-api-2.0.jar`is located in the `/EJBServer/components/Rest/restlib/dependencyLibsCore` directory.
- If a `ClassNotFoundException` occurs in your development environment, ensure that your J2EE_JAR eclipse classpath variable is correctly set to point to the correct and up to date Java EE JAR file. Older versions of the Java EE JAR file do not contain some of the classes that Jersey requires.
- If you are using an older version of an application server that uses a Java EE 6 JAR file, you must add more JAR files in addition to the ones in the list of required JARs. However, you might still encounter conflicts if your application server already contains JARs that use the JAX-RS 1.0 specification. Fixing these conflicts for older versions of application servers is outside the scope of this information. The following list outlines the JARS that are not

included in the Java EE 6 JAR and their locations. If your application server uses a Java EE 6 JAR file, you must also add these JAR files.

- `/EJBServer/components/Rest/restlib/dependencyLibsCore/javax.ws.rs-api-2.0.jar`
- `/EJBServer/components/Rest/restlib/dependencyLibsCore/custom.javax.annotation-api-<version>.jar`
- `/EJBServer/components/Rest/restlib/dependencyLibsCore/javax.inject-<version>.jar`

Adding client request filters

You can add a request filter to use as a hook point into the request flow and handle extra processing that applies to all outbound requests. Create the request filter and register it with the client to apply it to all outbound requests.

You can implement a request filter to invoke for all outbound requests before you send the request. For example, if you want to add an `Accept-Language` request header with the same value to all outbound requests, you can provide the code once in a filter. You do not need to add the code to each request.

Creating a client request filter

A request filter is a class that implements the `ClientRequestFilter` interface. The following example shows an implementation that adds the `Accept-Language` header and sets a value of `en-US`:

```
public class OutboundRequestFilter implements ClientRequestFilter {

    @Override
    public void filter(final ClientRequestContext requestContext) throws IOException {
        requestContext.getHeaders().add("Accept-Language", "en-US");
    }

}
```

Registering a client request filter with a client

Add the client request filter to all outbound requests made by a client by registering it with the client when you create it. The following code shows how to register the `OutboundRequestFilter`:

```
client.register(new OutboundRequestFilter());
```

Communicating over HTTPS/SSL

If communication with the API service is over HTTPS/SSL, the communication is handled by your application server.

Note: If you try to make an outbound API request to an API where the API provider uses a self-signed certificate, you might get a `SSLHandshakeException`. In a test or development

environment, if you want to proceed to call the API that uses a self-signed certificate, you can configure the Jersey REST client to accept any self-signed certs. Instructions for how to configure the Jersey client in this way are outside the scope of this information, but you can find them on the internet. Do not use this configuration in a production environment.

Notices

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the Merative website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of Merative

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of Merative.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

Merative reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by Merative, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

MERATIVE MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Merative or its licensors may have patents or pending patent applications covering subject matter described in this document. The furnishing of this documentation does not grant you any license to these patents.

Information concerning non-Merative products was obtained from the suppliers of those products, their published announcements or other publicly available sources. Merative has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-Merative products. Questions on the capabilities of non-Merative products should be addressed to the suppliers of those products.

Any references in this information to non-Merative websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those

websites are not part of the materials for this Merative product and use of those websites is at your own risk.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

The licensed program described in this document and all licensed material available for it are provided by Merative under terms of the Merative Client Agreement.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to Merative, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. Merative, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. Merative shall not be liable for any damages arising out of your use of the sample programs.

Privacy policy

The Merative privacy policy is available at https://www.merative.com/privacy.

Trademarks

Merative ™ and the Merative ™ logo are trademarks of Merative US L.P. in the United States and other countries.

IBM®, the IBM® logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

Adobe™, the Adobe™ logo, PostScript™, and the PostScript™ logo are either registered trademarks or trademarks of Adobe™ Systems Incorporated in the United States, and/or other countries.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft™, Windows™, and the Windows™ logo are trademarks of Microsoft™ Corporation in the United States, other countries, or both.

UNIX™ is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.