Service call type: AQ_RestServiceClient

The `AQ_RestServiceClient` service makes it possible to call a REST webservice.

Parameters

The following parameters can be set for the service from Encore:

Name	Description	Type	Required	Direction	Select module
restService	The name of the REST service along with the module where it is defined.	Module Element	Yes	Input	Yes
operation	The name of the operation of the REST service that needs to be called.	String	Yes	Input	No
url	The URL of the REST webservice.	String Expression	No	Input	No
connectionOverride	The name to use when overriding connection in the Runtime configuration files (more about this in the following section). Not that if the expression returns the value *unknown* (?) as result, an error is thrown. In this scenario no fallback is used.	String Expression	No	Input	No
mapping	The name of the data mapping to use along with the module where it can be found.	Data mapping	No	BiDirectional	No
username	The username to use in the service calling.	String	No	Input	No
password	The password to use in the service calling.	String	No	Input	No

Overriding parameters

The parameters url, username and password can be overridden in the Runtime's configuration files.

To do so, one can use either the name of the service call of type AQ_RestServiceClient in Blueriq Encore or the value given to the connectionOverride field.

The overriding can be done in the *application.properties* file by either using the name of the service call in Blueriq Encore (SendRest20) like this:

```
blueriq.connection.SendRest20.http.url=http://randomUrl:8080
blueriq.connection.SendRest20.http.username=randomUsername
blueriq.connection.SendRest20.http.password=randomPassword
```

... or by using the connection name (connectionName) like this:

```
blueriq.connection.connectionName.http.url=http://randomUrl:8080
blueriq.connection.connectionName.http.username=randomUsername
blueriq.connection.connectionName.http.password=randomPassword
```

In order to illustrate the benefit of using the new *connectionOverride* parameter a simple example will be presented.

Say there is webservice with 5 operations: *add, subtract, multiply, divide* and *modulo*. For each operation there is a corresponding service call and for each of those service calls the *url, username* and *password* are overridden in the *application.properties* file using the name of the service call. This means that even if the services call the same webservice (thus calling the same url with the same credentials) the override data must be specified for each of them. This is where the connectionOverride becomes useful by allowing each of those services to use a common connection name and eliminating the need of duplication.

If the value for one of those parameters is specified in more than one place (for instance if the ***url*** parameter is defined in Encore but also overridden using the service name) the following prioritization is used by the Runtime:

1. If there is a `connectionOverride` defined in Encore then that connection will be used over the others;
2. If there is no `connectionOverride` defined in Encore, the Runtime will look into the configuration files for (and will use if it exists) a property to override parameters using the service's name;
3. Finally, if there is no `connectionOveride` value defined in Encore and there is no property to override a parameter in the configuration files then the values defined in the model in Encore for the *url, username* and *password* will be used.

Final considerations

One must be consistent in the way it assigns value to the service parameters. Giving value to one parameter in a place and to another parameter in another place might prove faulty. For instance if one overrides the ***url*** by using a connection override but keeps the **username** and **password** values in Encore, the latter will not be used by the Runtime since it expects them to also be overridden in the configuration files.

Error handling

When the runtime sumbles upon an arror during execution of the AQ_RestServiceClient, there are serveral options to handle this:

- The exception exit can be used to handle exceptions such as a failing data mapping, or a response body that doesn't match the domain schema.
- Use the exit events to handle timeouts, client errors (http status code 4xx), and server errors (http status code 5xx).
    - If there is a need to distinguish on a more specific http status code than the aforemantioned ranges, you can use the header in the REST service to map the status code with the name "Status" to the profile.

Exit events

Name	Description	Type
Timeout	When the REST request returns a timeout exception.	Continue
ClientError (since 16.7)	When the REST request returns a 4xx exception.	Continue
ServerError (since 16.7)	When the REST request returns a 5xx exception.	Continue
default exit event	All unmapped events will be redirected to the *default* exit node of the service call, even errors. Therefore it is recommended to always map all possible expected exit events.	Continue

Authentication options

There are several authentication options, which are configurable per connection with properties:

No authentication

When no username and password are set for the service call (none in Encore and also none in the Connections Properties), then no authentication will be attempted. You should use this when the Rest Endpoint that you are calling is not secured.

Basic authentication

When a username and password are supplied, either in Encore or in the Connections Properties, these credentials will be supplied to the Rest Endpoint.

OpenID Connect authentication

See OpenID Connect for more information.

Oauth2 authentication

OAuth2 describes an authentication scheme. In its specification, it is explained **what** information should be in requests and responses, but not **how** this information should be transmitted.

Therefore, we use sensible defaults as described in this section. If you need other behavior, you can develop a custom Blueriq extension, as described below.

When the Rest Endpoint that needs to be called is secured with OAuth2, you can set the property `blueriq.connection.<connectionName>.http.authentication` to `oauth2`. You need to define a Spring Security Oauth2 Client Registration and Provider and set the `blueriq.connection.<connectionName>.http.oauth2-client-registration` to the corresponding client registration. See Connections Properties for those.

Since 17.0 we use Spring Security OAuth2, which makes it a lot more versatile and better configurable.

After

```
spring:
  security:
    oauth2:
      client:
        registration:
          my-oauth2-client:
            provider: my-auth-server
            client-id: my-client-id
                        client-authentication-method: client_secret_basic
            client-secret: secret-password-text
            authorization-grant-type: client_credentials
        provider:
          my-auth-server:
            token-uri: https://identity.provider.com/token
blueriq:
  connection:
    my-connection1:
      http:
        url: https://some.domain.com/resource1
        authentication: oauth2
        oauth2-client-registration: my-oauth2-client
    my-connection2:
      http:
        url: https://some.domain.com/resource2
        authentication: oauth2
        oauth2-client-registration: my-oauth2-client
```

- When requesting a token, the Client ID and the Client Secret will be sent as Basic Authentication as default, but you can also use client_secret_post as client-authentication-method so it will be sent in the body.
- When requesting a token, the POST method is used.
- In the token response, we expect a JSON structure that at least contains an `access_token` and a `token_type`:

  ```
  {
    "access_token": "f608a968-b1ef-457a-8d1a-71ee007ac4d2",
    "token_type": "bearer"
  }
  ```

- Access tokens are not cached. Each Rest Service call will request a new token.

Customization

If you need different behavior, you can write your own custom Access Token Provider, that needs to implement the `com.blueriq.component.api.oauth2.Oauth2AccessTokenProvider` interface and you need to define it as a Spring Bean in your application.

Limitations

- The current implementation is limited to one Access Token Provider per Blueriq Runtime, so all of your OAuth2 enabled Rest Service Calls will use the same implementation of the Access Token Provider.
- The default implementation only supports the `client_credentials` grant type.