Java Demo

Scott Hurrey

Demo using Java

The rest demo script demonstrates authenticating a REST application, management and use of the authorization token, and creating, updating, discovering, and deleting supported Learn objects. For a more complete tutorial, check out this page to walkthrough building a Java command-line application.

Prerequisites

- You must register a developer account and application in the Developer Portal
- You must register your application in Learn
- You must also configure the script as outlined in the README for the project

This webapp allows you to:

- Authenticate
- Create, Read, and Update a Data Source
- Create, Read, and Update a Term
- Create, Read, and Update a Course
- Create, Read, and Update a User
- Create, Read, and Update a Membership
- Delete created objects in reverse order of create - membership, user, course, term, datasource.

All generated output is sent to the browser.

This is not meant to be a Java tutorial. It will not teach you to write code in Java. It will, however, give a Developer familiar with Java the knowledge necessary to build a Web Services integration.

Assumptions

This help topic assumes the Developer:

- is familiar with Java
- has Tomcat running somewhere the webapp can be installed
- has obtained a copy of the source code and built and deployed it to Tomcat in conjunction with the project README.md file.
- has a REST-enabled Learn instance.

Code Walkthrough

To build an integration with the Learn REST Web Services, regardless of the programming language of choice, can really be summed up in two steps:

1. Use the Application Key and Secret to obtain an OAuth 2.0 access token, as described in the Basic Authentication document.
2. Call the appropriate REST endpoint with the appropriate data to perform the appropriate action.

Authorization and Authentication

The REST Services rely on OAuth 2.0 Bearer Tokens for authentication. A request is made to the token endpoint with a Basic Authorization header containing the base64-encoded key:secret string as its key. The token service returns a JSON object containing the Access Token, the Token Type, and the number of seconds until the token expires. The token is set to expire after one hour, and subsequent calls to retrieve the token will return the same token with an updated expiry time until such time that the token has expired. There is no refresh token and currently no revoke token method.

The java code handles this in `bbdn.rest.Authorizer`:

```
    HttpHeaders headers = new HttpHeaders();
    headers.add("Authorization", "Basic " + getHash());
    headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);

    HttpEntity<String> request = new
↪ HttpEntity<String>("grant_type=client_credentials",headers);

    ResponseEntity<Token> response = restTemplate.exchange(uri, HttpMethod.POST, request,
↪ Token.class);

    Token token = response.getBody();
```

The JSON response is serialized into the Token object, and you may then retrieve those values from that object.

Calling Services

The individual service calls are handled by Java Classes in service specific packages, that all implement the `bbdn.rest.RestHandler` interface. The interface is used to normalize each service handler to make additional service implementation standardized as new endpoints are added.

RestHandler dictates that four methods must be implemented:

- String createObject(String access_token);
- String readObject(String access_token);
- String updateObject(String access_token);
- String deleteObject(String access_token);

Each of these methods creates the JSON body when appropriate and then calls `bbdn.rest.RestRequest` to generate the appropriate HTTP Request, ship it to Learn, and return the JSON response as a String to be displayed in the browser window.

This all happens with the following code:

```
public static String sendRequest(String sUri, HttpMethod method, String access_token,
↪  String body) {
     try {
          RestTemplate restTemplate = new RestTemplate();
          // Workaround for allowing unsuccessful HTTP Errors to still print to the screen
          restTemplate.setErrorHandler(new DefaultResponseErrorHandler(){
              protected boolean hasError(HttpStatus statusCode) {
                    return false;
          }});
          // Workaround to allow for PATCH requests
          HttpComponentsClientHttpRequestFactory requestFactory = new
↪ HttpComponentsClientHttpRequestFactory();
          restTemplate.setRequestFactory(requestFactory);

          URI uri = null;
          try {
```

```
                uri = new URI(RestConstants.HOSTNAME + sUri);
            } catch (URISyntaxException e) {
                e.printStackTrace();
            }

            HttpHeaders headers = new HttpHeaders();
            headers.add("Authorization", "Bearer " + access_token);
            headers.setContentType(MediaType.APPLICATION_JSON);
            HttpEntity<String> request = new HttpEntity<String>(body, headers);

            ResponseEntity<String> response = restTemplate.exchange(uri, method, request,
↪ String.class);
            return (response.toString());
        }
        catch (Exception e) {
            return(e.getMessage());
        }
    }
```

End points are generally defined as `/learn/api/public/v1/<object type>/<objectId>`. Object ID can be either the pk1, like `_1_1`, or as the batchuid. This value should be prepended by externalId:, like `externalId:test101`.

For example, to retrieve a course by the pk1 `_1_1`, you would call **GET /learn/api/public/v1/courses/_1_1**. To retrieve by the batchuid `test101`, you would call **GET /learn/api/public/v1/courses/externalId:test101.**

Create is sent to Learn as a HTTP POST message with a JSON body that defines the object. The endpoint should omit the objectId, as this will be generated on creation.

Read is sent to Learn as a HTTP GET message with an empty body. The endpoint should include the objectId being retrieved.

Update is sent to Learn as a HTTP PATCH message with a JSON body that defines the object. The endpoint should include the objectId being updated.

Delete is sent to Learn as a HTTP DELETE message with empty body. The endpoint should include the objectId being deleted.

Datasources

Datasources are handled in `bbdn.rest.datasources.DatasourceHandler`. As illustrated above, this Class implements the RestHandler interface and exposes four methods. It also includes a private method to create the JSON payload.

Create

```
@Override
public String createObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.DATASOURCE_PATH, HttpMethod.POST,
↪ access_token, getBody()));
}
```

Read

```
@Override
public String readObject(String access_token) {
    return(RestRequest.sendRequest(RestConstants.DATASOURCE_PATH + "/externalId:" +
↪ RestConstants.DATASOURCE_ID, HttpMethod.GET, access_token, ""));
}
```

Update

```
@Override
public String updateObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.DATASOURCE_PATH + "/externalId:" +
↪  RestConstants.DATASOURCE_ID, HttpMethod.PATCH, access_token, getBody()));
}
```

Delete

```
 @Override
public String deleteObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.DATASOURCE_PATH + "/externalId:" +
↪  RestConstants.DATASOURCE_ID, HttpMethod.DELETE, access_token, ""));
}
```

Create Body

```
private String getBody() {
     ObjectMapper objMapper = new ObjectMapper();
     JsonNode datasource = objMapper.createObjectNode();
     ((ObjectNode) datasource).put("externalId", RestConstants.DATASOURCE_ID);
     ((ObjectNode) datasource).put("description", RestConstants.DATASOURCE_DESCRIPTION);
     String body = "";
     try {
          body = objMapper.writeValueAsString(datasource);
     } catch (JsonProcessingException e) {
          e.printStackTrace();
     }
     return(body);
}
```

Terms

Terms are handled in `bbdn.rest.terms.TermHandler`. As illustrated above, this Class implements the RestHandler interface and exposes four methods. It also includes a private method to create the JSON payload. In this initial release, we are omitting the datasource. This is because the externalId version of the datasource is not accepted in JSON payloads at this time. We could create a CONSTANT and set it to what we think it will be, but the ID isn't set until the Datasource is created, so we don't know for sure what it will be.

Create

```
@Override
public String createObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.TERM_PATH, HttpMethod.POST, access_token,
↪  getBody()));
}
```

Read

```
@Override
public String readObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.TERM_PATH + "/externalId:" +
↪  RestConstants.TERM_ID, HttpMethod.GET, access_token, ""));
}
```

Update

```
@Override
public String updateObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.TERM_PATH + "/externalId:" +
↪  RestConstants.TERM_ID, HttpMethod.PATCH, access_token, getBody()));
}
```

Delete

```
@Override
public String deleteObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.TERM_PATH + "/externalId:" +
↪  RestConstants.TERM_ID, HttpMethod.DELETE, access_token, ""));
}
```

Create Body

```
private String getBody() {
     ObjectMapper objMapper = new ObjectMapper();
     ObjectNode term = objMapper.createObjectNode();
     term.put("externalId", RestConstants.TERM_ID);
     //term.put("dataSourceId", RestConstants.DATASOURCE_ID);
     term.put("name", RestConstants.TERM_NAME);
     term.put("description", RestConstants.TERM_DISPLAY);
     ObjectNode availability = term.putObject("availability");
     availability.put("available", "Yes");
     ObjectNode duration = availability.putObject("duration");
     duration.put("type",  "Continuous");
     String body = "";
     try {
          body = objMapper.writeValueAsString(term);
     } catch (JsonProcessingException e) {
          e.printStackTrace();
     }
     return(body);
}
```

Courses

Course are handled in `bbdn.rest.course.CourseHandler`. As illustrated above, this Class implements the RestHandler interface and exposes four methods. It also includes a private method to create the JSON payload. In this initial release, we are omitting the datasource. This is because the externalId version of the datasource is not accepted in JSON payloads at this time. We could create a CONSTANT and set it to what we think it will be, but the ID isn't set until the Datasource is created, so we don't know for sure what it will be.

Create

```
@Override
public String createObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.COURSE_PATH, HttpMethod.POST,
↪  access_token, getBody()));
}
```

Read

```
@Override
public String readObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.COURSE_PATH + "/externalId:" +
↪  RestConstants.COURSE_ID, HttpMethod.GET, access_token, ""));
}
```

Update

```
@Override
public String updateObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.COURSE_PATH + "/externalId:" +
↪  RestConstants.COURSE_ID, HttpMethod.PATCH, access_token, getBody()));
```

```
}
```

Delete

```
@Override
public String deleteObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.COURSE_PATH + "/externalId:" +
↪  RestConstants.COURSE_ID, HttpMethod.DELETE, access_token, ""));
}
```

Create Body

```
private String getBody() {
     ObjectMapper objMapper = new ObjectMapper();
     ObjectNode course = objMapper.createObjectNode();
     course.put("externalId", RestConstants.COURSE_ID);
     //course.put("dataSourceId", RestConstants.DATASOURCE_ID);
     course.put("courseId", RestConstants.COURSE_ID);
     course.put("name", RestConstants.COURSE_NAME);
     course.put("description", RestConstants.COURSE_DESCRIPTION);
     course.put("allowGuests", "true");
     course.put("readOnly", "false");
     course.put("termId", RestConstants.TERM_ID);
     ObjectNode availability = course.putObject("availability");
     availability.put("duration", "continuous");
     String body = "";
     try {
          body = objMapper.writeValueAsString(course);
     } catch (JsonProcessingException e) {
          e.printStackTrace();
     }
     return(body);
}
```

Users

Users are handled in `bbdn.rest.users.UserHandler`. As illustrated above, this Class implements the RestHandler interface and exposes four methods. It also includes a private method to create the JSON payload. In this initial release, we are omitting the datasource. This is because the externalId version of the datasource is not accepted in JSON payloads at this time. We could create a CONSTANT and set it to what we think it will be, but the ID isn't set until the Datasource is created, so we don't know for sure what it will be.

Create

```
@Override
public String createObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.USER_PATH, HttpMethod.POST, access_token,
↪  getBody()));
}
```

Read

```
@Override
public String readObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.USER_PATH + "/externalId:" +
↪  RestConstants.USER_ID, HttpMethod.GET, access_token, ""));
}
```

Update

```
@Override
public String updateObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.USER_PATH + "/externalId:" +
↪  RestConstants.USER_ID, HttpMethod.PATCH, access_token, getBody()));
}
```

Delete

```
@Override
public String deleteObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.USER_PATH + "/externalId:" +
↪  RestConstants.USER_ID, HttpMethod.DELETE, access_token, ""));
}
```

Create Body

```
private String getBody() {
     ObjectMapper objMapper = new ObjectMapper();
     ObjectNode user = objMapper.createObjectNode();
     user.put("externalId", RestConstants.USER_ID);
     //user.put("dataSourceId", RestConstants.DATASOURCE_ID);
     user.put("userName", RestConstants.USER_NAME);
     user.put("password", RestConstants.USER_PASS);
     ObjectNode availability = user.putObject("availability");
     availability.put("available", "Yes");
     ObjectNode name = user.putObject("name");
     name.put("given", RestConstants.USER_FIRST);
     name.put("family", RestConstants.USER_LAST);
     ObjectNode contact = user.putObject("contact");
     contact.put("email", RestConstants.USER_EMAIL);
     String body = "";
     try {
          body = objMapper.writeValueAsString(user);
     } catch (JsonProcessingException e) {
          e.printStackTrace();
     }
      return(body);
}
```

Memberships

Memberships are handled in `bbdn.rest.memberships.MemberHandler`. As illustrated above, this Class implements the RestHandler interface and exposes four methods. It also includes a private method to create the JSON payload. In this initial release, we are omitting the datasource. This is because the externalId version of the datasource is not accepted in JSON payloads at this time. We could create a CONSTANT and set it to what we think it will be, but the ID isn't set until the Datasource is created, so we don't know for sure what it will be. In addition, the endpoint for memberships is a bit different, in that it is a sub-call to courses, so the endpoint would look like `/learn/api/public/v1/courses/<courseId>/users/<userId>`.

Create

```
@Override
public String createObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.COURSE_PATH + "/externalId:" +
↪  RestConstants.COURSE_ID + "/users/externalId:"+ RestConstants.USER_ID, HttpMethod.PUT,
↪  access_token, getBody()));
}
```

Read

```
@Override
public String readObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.COURSE_PATH + "/externalId:" +
↪  RestConstants.COURSE_ID + "/users/externalId:"+ RestConstants.USER_ID, HttpMethod.GET,
↪  access_token, ""));
}
```

Update

```
@Override
public String updateObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.COURSE_PATH + "/externalId:" +
↪  RestConstants.COURSE_ID + "/users/externalId:"+ RestConstants.USER_ID,
↪  HttpMethod.PATCH, access_token, getBody()));
}
```

Delete

```
@Override
public String deleteObject(String access_token) {
     return(RestRequest.sendRequest(RestConstants.COURSE_PATH + "/externalId:" +
↪  RestConstants.COURSE_ID + "/users/externalId:"+ RestConstants.USER_ID,
↪  HttpMethod.DELETE, access_token, ""));
}
```

Create Body

```
private String getBody() {
     ObjectMapper objMapper = new ObjectMapper();
     ObjectNode membership = objMapper.createObjectNode();
     //membership.put("dataSourceId", RestConstants.DATASOURCE_ID);
     ObjectNode availability = membership.putObject("availability");
     availability.put("available", "Yes");
     membership.put("courseRoleId", "Instructor");
     String body = "";
     try {
          body = objMapper.writeValueAsString(membership);
     } catch (JsonProcessingException e) {
          e.printStackTrace();
     }
     return(body);
}
```

Conclusion

All of the code snippets included in this document at included in a sample REST Demo Java Webapp application available on GitHub. There is a README.html included that talks more specifically about building and running the code. Feel free to review the code and run it against a test or development Learn instance to see how it works.