Consuming a RESTful web service

Explore how to access a simple RESTful web service and consume its resources in Java using JSON-B and JSON-P.

What you'll learn

You will learn how to access a REST service, serialize a Java object that contains a list of artists and their albums, and use two different approaches to deserialize the returned JSON resources. The first approach consists of using the JSON-B API to directly convert JSON messages into Java objects. The second approach consists of using the JSON-P API to process the JSON.
The REST service that provides the artists and albums resources has already been written for you and is accessible at:

```
http://localhost:9080/artists
```

Which responds with the following JSON:

```
[
    {
        "name" : "foo",
        "albums" : [
            {
                "title" : "album_one",
                "artist" : "foo",
                "ntracks" : 12
            },
            {
                "title" : "album_two",
                "artist" : "foo",
                "ntracks" : 15
            }
        ]
    },
```

```
    {
        "name" : "bar",
        "albums" : [
            {
                "title" : "foo walks into a bar",
                "artist" : "bar",
                "ntracks" : 12
            }
        ]
    },
    {
        "name" : "dj",
        "albums" : [
        ]
    }
]
```

You will implement the following two endpoints using the two deserialization approaches:

- `…/artists/total` to return the total number of artists in the JSON
- `…/artists/total/<artist>` to return the total number of albums in the JSON for the particular artist

Getting started

The fastest way to work through this guide is to clone the Git repository and use the projects that are provided inside:

```
git clone https://github.com/sunilake/guide-rest-client-java.git
cd guide-rest-client-java
```

The `start` directory contains the starting project that you will build upon.
The `finish` directory contains the finished project, which is what you will build.

Starting the service

This guide is already setup with a general application. As you progress through the guide you will make updates to the code directly, and then push updates to the server so you can see the results.

To start the REST service, run the Maven `install` and `liberty:start-server` goals from the `start`directory:

```
mvn clean install

mvn liberty:start-server
```

When the server is running, you can find your service at http://localhost:9080/artists
After you are done checking out the application, stop the Open Liberty server:

```
mvn liberty:stop-server
```

Creating POJOs

To deserialize a JSON message, start with creating Plain Old Java Objects (POJOs) that represent what is in the JSON and whose instance members map to the keys in the JSON. For the purpose of this guide, you are given two POJOs: `Artist` and `Album`.
The `Artist` object has two instance members `name` and `albums`, which map to the artist name and the collection of the albums they have written. The `Album` object represents a single object within the album collection, and contains three instance members `title`, `artist`, and `ntracks`, which map to the album title, the artist who wrote the album, and the number of tracks the album contains.
You can view the Artist POJO by navigating to the `src/main/java/io/openliberty/guides/consumingrest/model/Artist.java` file:

```
package io.openliberty.guides.consumingrest.model;

import javax.json.bind.annotation.JsonbCreator;
import javax.json.bind.annotation.JsonbProperty;
import javax.json.bind.annotation.JsonbTransient;

public class Artist {
    public String name;
    public Album albums[];

    //does not map to anything
    @JsonbTransient
    public boolean legendary = true;

    //default constructor can be defined
    public Artist() {
```

```
    }

    @JsonbCreator
    //or custom constructor can be used
    public Artist(
      @JsonbProperty("name") String name,
      @JsonbProperty("albums") Album albums[]) {

      this.name = name;
      this.albums = albums;
    }

    @Override
    public String toString() {
      return name + " wrote " + albums.length + " albums";
    }
}
```

You can view the Album POJO by navigating to the `src/main/java/io/openliberty/guides/consumingrest/model/Album.java` file:

```
package io.openliberty.guides.consumingrest.model;

import javax.json.bind.annotation.JsonbCreator;
import javax.json.bind.annotation.JsonbProperty;

public class Album {
    public String title;

    @JsonbProperty("artist")
    public String artistName;

    @JsonbProperty("ntracks")
    public int totalTracks;
```

```
    //default constructor can be defined
    public Album() {
    }

    @JsonbCreator
    //or custom constructor can be used
    public Album(
      @JsonbProperty("title") String title,
      @JsonbProperty("artist") String artistName,
      @JsonbProperty("ntracks") int totalTracks) {

      this.title = title;
      this.artistName = artistName;
      this.totalTracks = totalTracks;
    }

    @Override
    public String toString() {
      return "Album titled " + title + " by " + artistName +
        " contains " + totalTracks + " tracks";
    }
}
```

Introducing JSON-B and JSON-P

JSON-B is a feature introduced with Java EE 8 and strengthens Java support for JSON. With JSON-B you directly serialize and deserialize POJOs. This API gives you a variety of options for working with JSON resources.

In contrast, you need to use helper methods with JSON-P to process a JSON response. This tactic is more straightforward, but it can be cumbersome with more complex classes.

JSON-B is built on top of the existing JSON-P API. JSON-B can do everything that JSON-P can do and allows for more customization for serializing and deserializing.

Using JSON-B

JSON-B requires a POJO to have a public default no-argument constructor for deserialization and binding to work properly.

The JSON-B engine includes a set of default mapping rules, which can be run without any customization annotations or custom configuration. In some instances, you might find it useful to deserialize a JSON message with only certain fields, specific field names, or classes with custom constructors. In these cases, annotations are necessary and recommended:

- The `@JsonbProperty` annotation to map JSON keys to class instance members and vice versa. Without the use of this annotation, JSON-B will attempt to do POJO mapping, matching the keys in the JSON to the class instance members by name. If these names do not match, then getters and setters are matched to JSON keys and the instance members are set explicitly. If the getter and setter signatures do not match any key names, then deserialization fails. The Artist POJO does not require this annotation because all instance members match the JSON keys by name.
- The `@JsonbCreator` and `@JsonbProperty` annotations to annotate a custom constructor. These annotations are required for proper parameter substitution when a custom constructor is used.
- The `@JsonbTransient` annotation to define an object property that does not map to a JSON property. While the use of this annotation is good practice, it is only necessary for serialization.

Consuming the REST resource

The `Artist` and `Album` POJOs are ready for deserialization. To consume the JSON response from your REST service, create the `Consumer` class in the `src/main/java/io/openliberty/guides/consumingrest/Consumer.java` file:

```
package io.openliberty.guides.consumingrest;


import java.util.List;
import java.util.stream.Collectors;


import javax.json.JsonArray;
import javax.json.JsonObject;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
```

```
import javax.ws.rs.core.Response;

import io.openliberty.guides.consumingrest.model.Album;
import io.openliberty.guides.consumingrest.model.Artist;

public class Consumer {
    public static Artist[] consumeWithJsonb(String targetUrl) {
      Client client = ClientBuilder.newClient();

      Response response = client.target(targetUrl).request().get();
      Artist[] artists = response.readEntity(Artist[].class);

      response.close();
      client.close();

      return artists;
    }

    public static Artist[] consumeWithJsonp(String targetUrl) {
      Client client = ClientBuilder.newClient();

      Response response = client.target(targetUrl).request().get();
      JsonArray arr = response.readEntity(JsonArray.class);

      response.close();
      client.close();

      return Consumer.collectArtists(arr);
    }

    private static Artist[] collectArtists(JsonArray artistArr) {
      List<Artist> artists = artistArr.stream().map(artistJson -> {
        JsonArray albumArr = ((JsonObject) artistJson).getJsonArray("albums");
        Artist artist = new Artist(
          ((JsonObject) artistJson).getString("name"),
          Consumer.collectAlbums(albumArr));
```

```
        return artist;
      }).collect(Collectors.toList());

      return artists.toArray(new Artist[artists.size()]);
    }

    private static Album[] collectAlbums(JsonArray albumArr) {
      List<Album> albums = albumArr.stream().map(albumJson -> {
        Album album = new Album(
          ((JsonObject) albumJson).getString("title"),
          ((JsonObject) albumJson).getString("artist"),
          ((JsonObject) albumJson).getInt("ntracks") );
        return album;
      }).collect(Collectors.toList());

      return albums.toArray(new Album[albums.size()]);
    }

}
```

Next, recompile the application:

```
mvn compile
```

Processing JSON using JSON-B

JSON-B is a Java API that is used to serialize Java objects to JSON messages and vice versa.

Open Liberty's JSON-B feature on Maven Central includes the JSON-B provider through transitive dependencies. To include the JSON-B provider in your project, add the following dependency to your `pom.xml` file, which is already done for you:

```
<dependency>
   <groupId>io.openliberty.features</groupId>
   <artifactId>jsonb-1.0</artifactId>
   <type>esa</type>
   <scope>provided</scope>
```

```
</dependency>
```

The `consumeWithJsonb` method in the `Consumer` class makes a `GET` request to the running artist service and retrieves the JSON. Then binds the JSON into an `Artist` array, use the `Artist[]` entity type in the `readEntity` call.

Processing JSON using JSON-P

The `consumeWithJsonp` method makes in the `Consumer` class makes a `GET` request to the running artist service and retrieves the JSON. This method then uses the `collectArtists` and `collectAlbums` helper methods. These helper methods will parse the JSON and collect its objects into individual POJOs. Notice that you can use the custom constructors to create instances of `Artist` and `Album`.

Creating additional REST resources

Now that you can consume a JSON resource you can put that data to use. Copy the `ArtistResource` class below and replace the `ArtistResource` class in the `src/main/java/io/openliberty/guides/consumingrest/service/ArtistResource.java` file:

```
package io.openliberty.guides.consumingrest.service;

import javax.json.JsonArray;
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.UriInfo;

import io.openliberty.guides.consumingrest.model.Artist;
import io.openliberty.guides.consumingrest.Consumer;

@Path("artists")
```

```
public class ArtistResource {

    @Context
    UriInfo uriInfo;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public JsonArray getArtists() {
            return Reader.getArtists();
    }

    @GET
    @Path("jsonString")
    @Produces(MediaType.TEXT_PLAIN)
    public String getJsonString() {
      Jsonb jsonb = JsonbBuilder.create();

      Artist[] artists = Consumer.consumeWithJsonb(uriInfo.getBaseUri().toString()
+
        "artists");
      String result = jsonb.toJson(artists);

      return result;
    }

    @GET
    @Path("total/{artist}")
    @Produces(MediaType.TEXT_PLAIN)
    public int getTotalAlbums(@PathParam("artist") String artist) {
      Artist[] artists = Consumer.consumeWithJsonb(uriInfo.getBaseUri().toString()
        + "artists");

      for (int i = 0; i < artists.length; i++) {
        if (artists[i].name.equals(artist)) {
          return artists[i].albums.length;
        }
      }
```

```
      return -1;
    }

    @GET
    @Path("total")
    @Produces(MediaType.TEXT_PLAIN)
    public int getTotalArtists() {
      return Consumer.consumeWithJsonp(uriInfo.getBaseUri().toString() +
        "artists").length;
    }

}
```

- The `getArtists` method provides the raw JSON data service that you accessed at the beginning of this guide.
- The `getJsonString` method uses JSON-B to return the JSON as a string that will be used later for testing.
- The `getTotalAlbums` method uses JSON-B to return the total number of albums present in the JSON for a particular artist. The method returns -1 if this artist does not exist.
- The `getTotalArtists` method uses JSON-P to return the total number of artists present in the JSON.

The methods that you wrote in the `Consumer` class could be written directly in the `ArtistResource` class. However, if you are consuming a REST resource from a third party service, you should separate your `GET`/`POST` requests from your data consumption.
Next, recompile the application:

```
mvn compile
```

Testing deserialization

Create the `ConsumingRestTest` class in the `src/test/java/it/io/openliberty/guides/consumingrest/ConsumingRestTest.java` file:

```
package it.io.openliberty.guides.consumingrest;

import static org.junit.Assert.assertEquals;

import javax.json.bind.Jsonb;
```

```
import javax.json.bind.JsonbBuilder;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.core.Response;

import org.junit.After;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import io.openliberty.guides.consumingrest.model.Artist;

public class ConsumingRestTest {

    private static String port;
    private static String baseUrl;
    private static String targetUrl;

    private Client client;
    private Response response;

    @BeforeClass
    public static void oneTimeSetup() {
      port = System.getProperty("liberty.test.port");
      baseUrl = "http://localhost:" + port + "/artists/";
      targetUrl = baseUrl + "total/";
    }

    @Before
    public void setup() {
      client = ClientBuilder.newClient();
    }

    @After
    public void teardown() {
      client.close();
```

```
    }

    @Test
    public void testArtistDeserialization() {
      response = client.target(baseUrl + "jsonString").request().get();
      this.assertResponse(baseUrl + "jsonString", response);

      Jsonb jsonb = JsonbBuilder.create();

      String expectedString = "{\"name\":\"foo\",\"albums\":"
        + "[{\"title\":\"album_one\",\"artist\":\"foo\",\"ntracks\":12}]}";
      Artist expected = jsonb.fromJson(expectedString, Artist.class);

      String actualString = response.readEntity(String.class);
                  Artist[] actual = jsonb.fromJson(actualString, Artist[].class);

      assertEquals("Expected names of artists does not match", expected.name, actu
al[0].name);

      response.close();
    }

    @Test
    public void testJsonBAlbumCount() {
      String[] artists = {"dj", "bar", "foo"};
      for (int i = 0; i < artists.length; i++) {
        response = client.target(targetUrl + artists[i]).request().get();
        this.assertResponse(targetUrl + artists[i], response);

        int expected = i;
        int actual = response.readEntity(int.class);
        assertEquals("Album count for " + artists[i] + " does not match", expected
, actual);

        response.close();
      }
    }
```

```
    @Test
    public void testJsonBAlbumCountForUnknownArtist() {
      response = client.target(targetUrl + "unknown-artist").request().get();

      int expected = -1;
      int actual = response.readEntity(int.class);
      assertEquals("Unknown artist must have -1 albums", expected, actual);

      response.close();
    }

    @Test
    public void testJsonPArtistCount() {
      response = client.target(targetUrl).request().get();
      this.assertResponse(targetUrl, response);

      int expected = 3;
      int actual = response.readEntity(int.class);
      assertEquals("Expected number of artists does not match", expected, actual);

      response.close();
    }

    /**
     * Asserts that the given URL has the correct (200) response code.
     */
    private void assertResponse(String url, Response response) {
      assertEquals("Incorrect response code from " + url, 200, response.getStatus(
));
    }
}
```

Maven finds and executes all tests under `it/` and each test method must be marked with the `@Test`annotation.

You can use the `@BeforeClass` and `@AfterClass` annotations to perform any one time setup and teardown tasks before and after all of your tests execute, as well as the `@Before` and `@After` annotations to do the same but for each individual test case.
Next, recompile the application:

```
mvn compile
```

Testing the binding process

For your test classes to have access to JSON-B, add the following dependency to your `pom.xml` file, which is already done for you:

```
<dependency>
    <groupId>org.eclipse</groupId>
    <artifactId>yasson</artifactId>
    <version>1.0.1</version>
    <scope>test</scope>
</dependency>
```

The `testArtistDeserialization` test case checks that `Artist` instances created from the REST data and those that are hardcoded perform the same.
The `assertResponse` helper method ensures that the response code you receive is valid (200).

Processing with JSON-B test

The `testJsonBAlbumCount` and `testJsonBAlbumCountForUnknownArtist` tests both use the `total/{artist}` endpoint which invokes JSON-B.
The `testJsonBAlbumCount` test case checks that deserialization with JSON-B was done correctly and that the correct number of albums is returned for each artist in the JSON.
The `testJsonBAlbumCountForUnknownArtist` test case is similar to `testJsonBAlbumCount` but instead checks an artist that does not exist in the JSON and ensures that a value of '-1' is returned.

Processing with JSON-P test

The `testJsonPArtistCount` test uses the `total` endpoint which invokes JSON-P. This test checks that deserialization with JSON-P was done correctly and that the correct number of artists is returned.

Running the tests

If the server is still running from the previous steps, stop it using the Maven `liberty:stop-server` goal from command line in the `start` directory:

```
mvn liberty:stop-server
```

Then, verify that the tests pass using the Maven `verify` goal:

```
mvn verify
```

It may take some time before build is complete. If the tests pass, you will see a similar output to the following:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running it.io.openliberty.guides.consumingrest.ConsumingRestTest
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.59 sec - in it.io.openliberty.guides.consumingrest.ConsumingRestTest

Results :

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
```

Great work! You're done!

You have just accessed a simple RESTful web service and consumed its resources using JSON-B and JSON-P.