CS 548—Fall 2022
Enterprise Software Architecture and Design
Assignment Six—REST Web Service

In the previous assignment, you developed a Web application that allowed the domain model to be queried. In this assignment, you define a Web service that allows new patients, providers and treatments to be added to the domain model, as well as allowing programmatic querying of the domain model. You will use a Web service client application to upload data using this Web service.

Web Service

The Web service is defined in a JAX-RS project, `clinic-rest`. There are two resources in this Web service:
1. Patient: This resource allows patient representations to be downloaded, as patient DTOs.
2. Provider: This resource allows provider and treatment representations to be downloaded. It also allows new patient, provider and treatment information to be uploaded.

The configuration of the Web service is given by this class:

```
@ApplicationPath("/")
public class AppConfig extends Application {
    ...
}
```

The context root of the Web service is defined in the `payara-web.xml` descriptor in the `WEB-INF` folder:

```
<payara-web-app error-url="">
    <context-root>/api</context-root>
</payara-web-app>
```

Each of the two resource classes is defined as (request-scoped) CDI beans, annotated as JAX-RS beans. They should define the following API for the Web service:

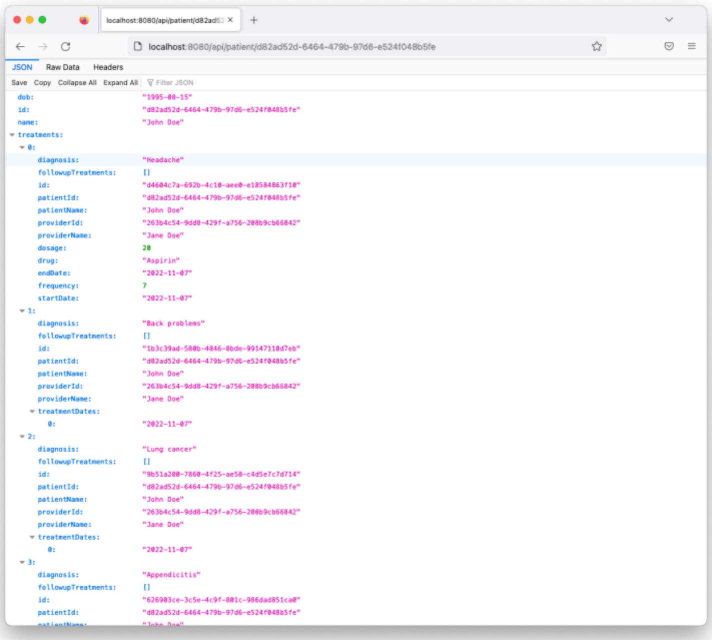
1. Obtaining a single patient representation, given a patient resource URI. An HTTP response code of 404 ("Not found") occurs if there is no patient for the specified URI. To support connectedness of hypermedia, the response should also provide links (URIs) to the treatments received by that patient, in the response headers.
   `GET /api/patient/`*`patient-id`*
2. Obtaining a single provider representation, given a provider resource URI. An HTTP response code of 404 ("Not found") occurs if there is no provider for the specified URI. To support connectedness of hypermedia, the response should also provide links (URIs) to the treatments administered by that provider, in the response headers.
   `GET /api/provider/`*`provider-id`*
3. Obtaining a single treatment representation, given a treatment resource URI. The treatment resource URI includes an identifier both for the treatment and for the

provider administering that treatment. An HTTP response code of 404 ("Not found") occurs if there is no provider or treatment for the specified URI. To support connectedness of hypermedia, the response should also provide links (URIs) to the patient receiving the treatment and the provider administering the treatment, in the response headers.
`GET /api/provider/`*`provider-id`*`/treatment/`*`treatment-id`*

4. Adding patients, providers and treatments. The operation for doing this expects a JSON object that includes three lists of DTOs, for patients, providers and treatments. Think of this as supporting a mobile app that captures this kind of information and periodically uploads it to the cloud[1]. Since the amount of data is unbounded, the operation implementing this should stream the input data, manually parsing the JSON data and deserializing the DTOs one by one, as you did in an earlier assignment for reading JSON data from a file.
`POST /api/provider`

The resource classes should be declared as request-scoped beans. The Web service project that should have a `beans.xml` file, in the `WEB-INF` directory, to enable CDI discovery of the bean classes. Within a resource class, use the `@Context` annotation to inject parts of the HTTP request context, e.g., the URI context that you will need to construct hyperlinks for your representations[2]. Use dependency injection (specifically CDI, using `@Inject`) to inject service beans that perform the actual service logic of performing the insertion, query and update operations.

You can test the GET methods of the Web service using a HTTP client like curl or postman, or you can use a Web browser:


localhost:8080/api/patient/d82ad52
localhost:8080/api/patient/d82ad52d-6464-479b-97d6-e524f048b5fe
JSON
Raw Data
Headers
Save
Copy
Collapse All
Expand All
Filter JSON

```
dob:                      "1995-08-15"
id:                       "d82ad52d-6464-479b-97d6-e524f048b5fe"
name:                     "John Doe"
▾ treatments:
  ▾ 0:
      diagnosis:          "Headache"
      followupTreatments: []
      id:                 "d4604c7a-692b-4c10-aee0-e18584863f10"
      patientId:          "d82ad52d-6464-479b-97d6-e524f048b5fe"
      patientName:        "John Doe"
      providerId:         "263b4c54-9dd8-429f-a756-208b9cb66842"
      providerName:       "Jane Doe"
      dosage:             20
      drug:               "Aspirin"
      endDate:            "2022-11-07"
      frequency:          7
      startDate:          "2022-11-07"
  ▾ 1:
      diagnosis:          "Back problems"
      followupTreatments: []
      id:                 "1b3c39ad-580b-4846-8bde-99147110d7eb"
      patientId:          "d82ad52d-6464-479b-97d6-e524f048b5fe"
      patientName:        "John Doe"
      providerId:         "263b4c54-9dd8-429f-a756-208b9cb66842"
      providerName:       "Jane Doe"
    ▾ treatmentDates:
        0:                "2022-11-07"
  ▾ 2:
      diagnosis:          "Lung cancer"
      followupTreatments: []
      id:                 "9b51a200-7860-4f25-ae58-c4d5e7c7d714"
      patientId:          "d82ad52d-6464-479b-97d6-e524f048b5fe"
      patientName:        "John Doe"
      providerId:         "263b4c54-9dd8-429f-a756-208b9cb66842"
      providerName:       "Jane Doe"
    ▾ treatmentDates:
        0:                "2022-11-07"
  ▾ 3:
      diagnosis:          "Appendicitis"
      followupTreatments: []
      id:                 "626903ce-3c5e-4c9f-801c-986dad851ca0"
      patientId:          "d82ad52d-6464-479b-97d6-e524f048b5fe"
```

[1] In a real application, the mobile app would synchronize with the cloud, uploading its data and downloading data already uploaded by other apps. We only consider uploading for this assignment.

You can retrieve the WADL description of the service, available at:
`http://host-name:8080/api/application.wadl`

Although Jakarta EE and Micro Profile define the JSON-B standard for serializing and deserializing data as JSON, this Web service uses Gson. See the `GsonProvider` class, that defines message body consumers and producers for converting Java objects to and from JSON.

Client Application

You are provided with a client application, `clinic-rest-client`, that you should use with this Web service, to upload the data. It is an adaptation of a command-line app from an earlier assignment, for saving clinic data into a file as JSON and loading the data as JSON from a file. This app adds an additional `upload` operation, for uploading to the server. You can specify the server URI on the command line, but there is also a default specified in a properties file:

```
java -jar clinic-rest-client.jar --server http://...:8080/api/
```

The app uses the Retrofit library to do a Web service call, using the okHttp client library for HTTP, and using Gson to serialize data as JSON. The entry point for performing the Web service call is a `WebClient` object. Given the base URI for the Web service, it creates a Retrofit client stub combining this base URL, an okHttp client stub and a Gson data converter (obtained from the earlier `GsonFactory`). The Retrofit client stub uses the `IServer` interface as the API for the Web service (where the annotations are Retrofit annotations rather than JAX-RS):

```
public interface IServerApi {
    @POST("provider")
    public Call<Void> upload(@Body RequestBody requestBody);
}
```

The `WebClient` object uses the Retrofit stub to implement this operation:

```
public void upload(final IStreamingOutput output) throws IOException { ...
```

where the argument type is specified as:

```
public interface IStreamingOutput {
    public void write(OutputStream out) throws IOException;
}
```

When a network connection is established with the Web service, the streaming output argument is called to provide the data that is uploaded as the request body to the service. Having the Web client call back into the main program, to perform the streaming of data, maintains a separation between the business logic of the Web service operations and the low-level details of the Web service call. The `upload` operation in `WebClient` is provided with a streaming output lambda when it is called to perform the upload of the data in the app:

```
IStreamingOutput output = new IStreamingOutput() {
    public void write(OutputStream output) throws IOException {
            try (JsonWriter wr = gson.newJsonWriter(new BufferedWriter(
                                            new OutputStreamWriter(output)))) {
                wr.beginObject();
                ...
                wr.endObject();
            }
        }
};
client = new WebClient(serverUri);
client.upload(output);
```

Submission

In addition to the classes from the previous assignment, this assignment requires the addition of two additional projects: `clinic-rest` and `clinic-rest-client` (and an updated `clinic-root`). Maven will generate three applications from these projects:

1. The Web application, `clinic-webapp.war`, unchanged from the previous assignment.
2. The Web service for this assignment, `clinic-rest.war` (another Web archive file, deployed in Payara as you have already done with `clinic-webapp.war`).
3. The standalone application for the Web service client, `clinic-rest-client.jar`. See above how to deploy it.

Your solutions should be developed for Payara 5.2022 (Jakarta EE 8). In addition, record short mpeg videos of a demonstration of your application being deployed and used. Show the output of the Web service operations for querying for patients, providers and treatments, including the links to related resources in the response headers. Use the Web app to show the contents of the database before and after uploading data using the Web service client; use the `list` command in the app to list the data before uploading. Make sure that your name appears at the beginning of the video. *Do not provide private information such as your email or cwid in the video.*

Your solution should be uploaded via the Canvas classroom. Your solution should consist of a zip archive with one folder, identified by your name. Within that folder, you should have **all** of the Eclipse projects, including the projects for DTOs, the domain model, the service and the Web app (and of the course the Web service and client). You should also provide the WADL file for your deployment, and videos demonstrating the working of your assignment. Finally, the root folder should contain the three jar/war files for your applications: `clinic-webapp.war`, `clinic-rest.war` and `clinic-rest-client.jar`.

As part of your submission, export your Eclipse project to your file system, and then include that folder as part of your archive file. You should also include the WADL description for your REST Web service. You can obtain the latter by deploying the application in Payara and pointing your browser at the URL for the RESTful application. Finally include demo videos as described above.