# CS 548—Fall 2023
# 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 uploaded and downloaded, as patient DTOs.
2. Provider: This resource allows provider and treatment representations to be downloaded. It also allows new provider and treatment information to be uploaded.

The configuration of the Web service is given by this class[1]:

```
@ApplicationPath("/")
public class AppConfig extends ResourceConfig {

   public AppConfig() {
      packages("edu.stevens.cs548.clinic.rest")
         .register(JsonGsonFeature.class);
   }
}
```

`ResourceConfig` is a subclass of the standard JAX-RS `Application` class, with additional functionality added by the Jersey implementation of JAX-RS that Payara uses. The `package` operation scans all classes in the specified package for resource classes, so they do not have to be listed individually. It also allows extensions (features) to be added at runtime; in this case, we add a feature for serializing data using Gson. We also implement a provider for Gson objects, that uses the factory we defined in an earlier assignment to provide a Gson customized to our application:

```
@Provider
public class GsonContextResolver implements ContextResolver<Gson> {
    @Override
    public Gson getContext(Class<?> aClass) {
        return GsonFactory.createGson();
    }
}
```

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

---

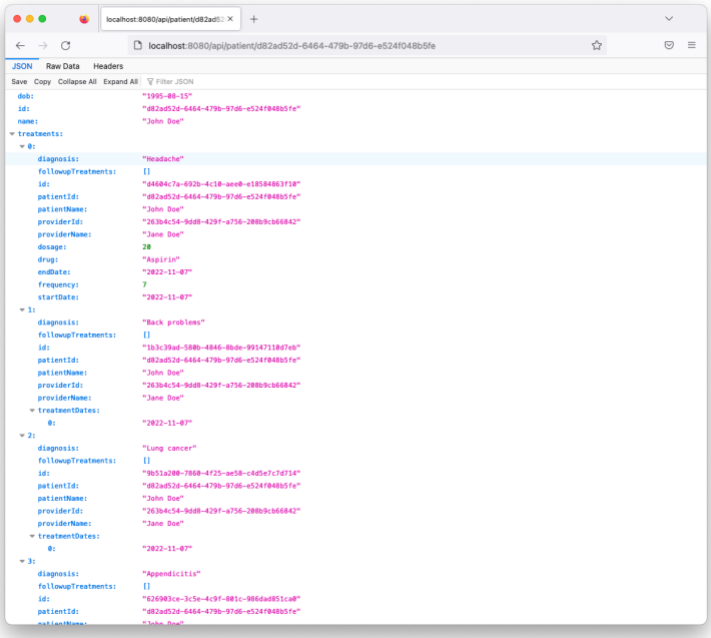
[1] You will also need to add the annotation for configuring the database connection, as you did for the Web application in the previous two assignments.

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. Adding a single patient to the resource of all patients. The body of the request should be a representation for a patient (as a JSON object).
   `POST /api/patient`
3. 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`
4. Adding a single provider to the resource of all providers. The body of the request should be a representation for a provider (as a JSON object).
   `POST /api/provider`
5. 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`
6. Adding a single provider to the resource of all providers. The body of the request should be a representation for a provider (as a JSON object). The treatment resource URI includes an identifier for the provider administering that treatment.
   `POST /api/provider/provider-id/treatment`

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[2] to inject parts of the HTTP request context, e.g., the URI context that you will need to construct hyperlinks for your representations[3]. 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:

[2] The Jersey framework is using the HK2 dependency injection framework, rather than CDI, to inject parts of the HTTP request context.




You can retrieve the WADL description of the service, available at:

`http://host-name:9090/api/application.wadl`

See below for more information about deploying the Web service.

## Client Application

You are provided with a client application, `clinic-rest-client`, that you should use with this Web service, to upload data. It is an adaptation of a command-line app from an earlier assignment, for saving clinic data into a background database. This app instead performs Web service calls to save the data in a Web application. 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://...:9090/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("patient")
    public Call<Void> addPatient(@Body PatientDto patientDto);

    @POST("provider")
    public Call<Void> addProvider(@Body ProviderDto providerDto);
```

```
    @POST("provider/{id}/treatment")
    public Call<Void> addTreatment(@Path("id") String providerId,
                                   @Body TreatmentDto treatmentDto);

}
```

The `WebClient` object uses the Retrofit stub to implement the operations for adding patients, providers and treatments, for example:

```
private static final String LOCATION = "Location";

private IServerApi client;

public URI addPatient(PatientDto patientDto) throws IOException {
    Response<Void> response = client.addPatient(patientDto).execute();
    if (response.isSuccessful()) {
        return URI.create(response.headers().get(LOCATION));
    } else {
        throw new IOException(...response.code());
    }
}
```

## Deployment

As with the previous Web application, you have two options for deploying your Web service[4]. You must first ensure that you have Postgresql running in a container (again, in EC2 or on your laptop, depending on where you are doing your development), exposing the port 5432 that both the application and Intellij IDEA will need to access the database server.

### Containerized Deployment Using Payara Micro

On your laptop or on EC2, create a directory called `cs548-rest`, move the `clinic-rest.war` file there, and navigate into this directory:

```
$ mkdir cs548-rest
$ mv clinic-rest.war cs548-rest
$ cd cs548-rest
```

In this directory, create a file called `Dockerfile` that copies the WAR file for the application to the container file system[5]:

```
FROM payara/micro:6.2023.9-jdk17
COPY --chown=payara:payara clinic-rest.war ${DEPLOY_DIR}
CMD [ "--contextroot", "api",
      "--deploy", "/opt/payara/deployments/clinic-rest.war" ]
ENV JVM_ARGS="--add-opens=java.base/java.io=ALL-UNNAMED"
```

Save this file, and create a custom server image:

---

[4] When testing, start the Web application **before** you add data using the Web service, since the Web application clears the database every time it is deployed. Also, you will need to bind to a different HTTP port than the Payara Micro default (8080), since the Web application will bind to that port first.

[5] There are four lines in the dockerfile , the third line is broken to fit in this document.

```
$ docker build -t cs548/clinic-rest .
$ docker images
$ cd ..
```

Create and start the server container, on the same virtual network as the database. Note that you specified the same image name as above when you executed docker build; this image will have been cached locally. You will need to expose several ports to allow access from your browser:

```
$ docker run -d --name clinic-rest --network cs548-network -p 9090:8080 -e
DATABASE_USERNAME=cs548user -e DATABASE_PASSWORD=YYYYYY -e DATABASE=cs548 -e
DATABASE_HOST=cs548db cs548/clinic-rest
```

### Native Deployment Using Payara Micro

If you are unable to deploy Payara Micro in a container (because you are developing on your laptop and there is no Payara Micro image available for that architecture), you can deploy the application natively (for now) on your laptop. Download the payara micro jar file[6], say to a file called `payara-micro.jar`, and then (in a directory that contains the jar files for both Payara Micro and the Web service app), run the application as follows (on MacOS and Linux)[7]:

```
$ export DATABASE_USERNAME="cs548user"
$ export DATABASE_PASSWORD="YYYYYY"
$ export DATABASE="cs548"
$ export DATABASE_HOST="localhost"
$ java --add-opens=java.base/java.io=ALL-UNNAMED -jar payara-micro.jar
    --deploy clinic-rest.war --contextroot api --port 9090
```

On Windows, run the web app as follows:

```
> set DATABASE_USERNAME="cs548user"
> set DATABASE_PASSWORD="YYYYYY"
> set DATABASE="cs548"
> set DATABASE_HOST="localhost"
> java --add-opens=java.base/java.io=ALL-UNNAMED -jar payara-micro.jar
    --deploy clinic-rest.war --contextroot api --port 9090
```

You can run the database server that you installed on EC2 in the first assignment, or you can install it in the same way on your laptop so you can develop locally there. The above code for running Payara Micro natively assumes the database server is running on the same machine as the web app, specifying `localhost` for the value of `DATABASE_HOST`. If the database server is running in EC2, you will need to replace `localhost` with the external IP address of the EC2 instance.

You should see output like the following in the logs once you have successfully deployed the Web service:

---

[6] https://www.payara.fish/downloads/payara-platform-community-edition/

[7] You may also want to redirect output to a file, so you can examine it with an editor. See the first assignment spec.

```
Payara Micro URLs:
http://...:9090/api

'clinic-rest' REST Endpoints:
GET  /api/application.wadl
POST /api/patient
GET  /api/patient/{id}
POST /api/provider
GET  /api/provider/{id}
POST /api/provider/{id}/treatment
GET   /api/provider/{id}/treatment/{tid}
```

## 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 using Payara Micro 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 run it.

Your solutions should be developed for Payara 6.2023 (Jakarta EE 10, Java 17). 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. 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 Maven 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, include the folder for the Intellij IDEA project 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 and pointing your browser at the URL (see above). Finally include demo videos as described above and in the rubric.**