CS 548—Fall 2017
Enterprise Software Architecture and Design
Assignment Seven—REST Web Services

In this assignment, you define a REST interface for the domain-driven design that you developed in an earlier assignment, as an alternative to the SOA architecture of the previous assignment. You will define resources that provide access to your domain model in terms of representations transferred to and from clients. You will define a Web service interface for clients that want to access your application over the Web, that includes a hypermedia network linking resources to other resources.

The Maven POM file for the ClinicApp module lists the components in the enterprise app that is deployed. The list of enterprise-specific artifacts listed in the POM file now expands to include those artifacts added by this assignment:

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-ear-plugin</artifactId>
   <configuration>
      <defaultLibBundleDir>lib</defaultLibBundleDir>
      <modules>
         <webModule>
            <groupId>edu.stevens.cs548</groupId>
            <artifactId>ClinicSoapWebService</artifactId>
            <contextRoot>/clinic-soap</contextRoot>
         </webModule>
         <webModule>
            <groupId>edu.stevens.cs548</groupId>
            <artifactId>ClinicRestWebService</artifactId>
            <contextRoot>/clinic-rest</contextRoot>
         </webModule>
         <ejbModule>
            <groupId>edu.stevens.cs548</groupId>
            <artifactId>ClinicService</artifactId>
         </ejbModule>
         <ejbModule>
            <groupId>edu.stevens.cs548</groupId>
            <artifactId>ClinicInit</artifactId>
         </ejbModule>
      </modules>
   </configuration>
</plugin>
```

Representations

Define the representations for your ROA. There are resources for patients and providers, as well as sub-resources for treatments. *The patient and provider representations include links to the treatment resources to which they are related.* Each treatment representation

links to the associated patient and provider resources. There are three different forms of treatment representations, one for each form of treatment (surgery, drug treatment, radiology). Assume that the service can provide two forms of representation for each of these treatment representations, either as XML or JSON documents. You should use JAXB to actually generate the representations. To do this, define XML Schemas for your representations, and then generate JAXB-annotated classes from those schemas. You will want treatment representations to link to their related patient and provider resources, and you will want patient and provider representations to provide an operation `getLinks()` that returns a list of the links to treatment resources hyperlinked from those representations. One way to do this is by adding behavior to representations for returning this list of links. See http://jaxb.java.net/guide/Adding_behaviors.html for information on how to add behaviors to a class generated by JAXB from an XML schema. In summary, your treatment representations will inherit from the JAXB-generated treatment classes, adding behaviors to the JAXB classes:

```
public class DrugTreatmentRepresentation extends DrugTreatmentType { … }
public class SurgeryRepresentation extends SurgeryType { … }
public class RadiologyRepresentation extends RadiologyType { … }
```

To construct a representation as part of a returned result from a Web service, you will want to define a constructor that builds representations from DTOs, for example:

```
public PatientRepresentation (PatientDto dto, UriInfo uriInfo) {
	this();
	this.id = getPatientLink(dto.getId(), uriInfo);
	this.patientId =  dto.getPatientId();
	this.name = dto.getName();
	this.dob = dto.getDob();
	this.age = PatientFactory.computeAge(dto.getDob());
	/*
	 * Call getTreatments to initialize empty list.
	 */
	List<LinkType> links = this.getTreatments();
	for (long t : dto.getTreatments()) {
		links.add(TreatmentRepresentation.getTreatmentLink(t, uriInfo));
	}
}
```

Similarly, when a representation is provided as an argument to a Web service operation (such as adding a patient or provider), you will want to extract the DTO from the representation before calling into the Web service. The main difference between DTOs and representations is that the latter have links to related resources (e.g., a treatment has hyperlinks to the related patient and provider resources). Therefore the constructor for a representation takes an argument of type `UriInfo`, in addition to the DTO, to translate entity identifiers to links.

Resources and Business Objects

Define a RESTful Web service API for the following patient operations:

1. Adding a patient to the clinic. The operation takes a patient representation (with no links to treatments) and returns a new patient URI in the response header. The HTTP response code should be 201 ("Created"), with the URI for the new patient resource in the `Location` response header.
   `POST /clinic-rest/resources/patient`
2. 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. Remember that a patient representation should provide links (URIs) to the treatments for that patient.
   `GET /clinic-rest/resources/patient/id`
3. Obtaining a single patient representation, given a patient identifier. An HTTP response code of 404 ("Not found") occurs if there is no patient for the specified patient identifier. Remember that a patient representation should provide links (URIs) to the treatments for that patient.
   `GET /clinic-rest/resources/patient/byPatientId?id=id`
4. Obtaining a single treatment representation, from a treatment sub-resource of a patient, given a treatment resource URI. An HTTP response code of 404 ("Not found") occurs if there is no patient for the specified patient identifier. Remember that a patient representation should provide links (URIs) to the treatments for that patient.
   `GET /clinic-rest/resources/patient/id/treatments/tid`

The provider operations are as follows:

1. Adding a provider to the clinic. The operation takes a provider representation (with no links to treatments) and returns a new provider URI in the response header. The HTTP response code should be 201 ("Created"), with the URI for the new provider resource in the `Location` response header. Remember that the provider representation should contain links (URIs) to the treatments supervised by this provider.
   `POST /clinic-rest/resources/provider`
2. Obtaining a single provider representation, given a provider NPI.
   `GET /clinic-rest/resources/provider/byNPI?id=npi`
   An HTTP response code of 404 ("Not found") occurs if there is no provider for the specified NPI.
3. Obtaining a single provider representation, given a provider key.
   `GET /clinic-rest/resources/provider/id`
   An HTTP response code of 404 ("Not found") occurs if there is no provider for the specified URI.
4. Adding a treatment for a patient treated by this provider:
   `POST /clinic-rest/resources/provider/id/treatments`
5. Getting a treatment administered by this provider:
   `GET /clinic-rest/resources/provider/id/treatments/tid`

As with the SOAP-based Web service, define your RESTful Web service in the Web tier, as JAX-RS resource classes, and use dependency injection (specifically CDI, using `@Inject`) to inject EJBs that performs the actual service logic of performing the insertion, query and update operations.

The resource classes should be declared as `@RequestScoped` beans. The EJB project that declares the EJBs injected into your resources should have a `beans.xml` file, in the `META-INF` directory, to enable CDI discovery of the bean classes. You should **not** declare the resource classes as EJB session beans. Within a resource class, use the `@Context` annotation (from the HK2 dependency injection framework) to inject parts of the HTTP request context, e.g., the URI context that you will need to construct hyperlinks for your representations[1].

Your representation classes should be JAXB classes, defined in a separate (JAXB) project. You should delineate links to other resources from data, by defining a separate namespace for these DAP links. See the ClinicRepresentation project for more information. Alternatively you could use an extended feature available in Jersey, the implementation of REST used in Glassfish, called declarative linking. But this would require bundling those Jersey extension libraries in your application, which could cause serious deployment issues for you, so you are advised not to bother.

You do not have to test the REST service from an external client for this assignment, but you should at least deploy it so that you can retrieve the WADL description of the service, available at:
`http://host-name:8080/clinic-rest/resources/application.wadl`

Submission

In addition to the classes from previous assignments, this assignment requires the development of two additional projects: `ClinicRepresentations` (for your representation classes), and `ClinicRestWebService` (a dynamic Web project that contains the definition of your REST Web service). You should add these additional projects to the deployment assembly for your enterprise archive project, `ClinicApp`.

Your solutions should be developed for Java 8 and Java EE 7. You should use Payara (Glassfish 4.1) and Eclipselink 2.5.x. In addition, record short mpeg or Quicktime videos of a demonstration of your application being deployed and of viewing the resulting `application.wadl` file. 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 at least these Eclipse projects: `ClinicRoot`, `ClinicApp`, `ClinicDomain`, `ClinicInit`, `ClinicDTOs`, `ClinicService`, `ClinicSoapWebService`, `ClinicRepresentations` and `ClinicRestWebService`. You should also provide a report in the root folder, called `README.pdf`, that contains a report on your solution, as well as the WADL file for your deployment, and videos demonstrating the working of your assignment (unless the videos were uploaded to Google Drive instead). Finally the root folder should contain the enterprise archive file (`ClinicApp.ear`) that you used to

[1] The dependency injection framework used by Jersey, the JAX-RS reference implementation used by Glassfish, is HK2 rather than CDI. However it should not be necessary to explicitly bridge these frameworks, although it might well be necessary in a more ambitious project.

deploy your application.

It is important that you provide a document that documents your submission, included as a Word, RTF or PDF document in your submission root folder. Name this document README.pdf. As part of your submission, export your Eclipse project to your file system, and then include that folder as part of your archive file. Your written report should include, in an appendix, the WADL description for your REST Web service. You can obtain the latter by deploying the application in Glassfish and pointing your browser at the URL for the RESTful application.