CS 548—Fall 2022
Enterprise Software Architecture and Design
Assignment Five—Service-Oriented Architecture

In this assignment, you define a SOA for the domain-driven design that you developed in the previous assignment. You will use the definition of data transfer objects (DTOs) from the second assignment in defining the API for the. You will define service façades that provide access to your domain model in terms of DTOs transferred to and from clients. These service façades are injected into a Web application that you will use to demonstrate your working code.

Your application consists of the following Maven projects:

1. `clinic-root`: The parent module that tracks the dependencies between the other modules and provides common definitions such as versions of dependencies.
2. `clinic-domain`: The domain model, including the DAOs and entity classes.
3. `clinic-dto`: The definition of the DTOs.
4. `clinic-service-client`: The interfaces for the service façades.
5. `clinic-service`: The implementations of the service façades.
6. `clinic-init`: A startup bean for initializing the database.
7. `clinic-webapp`: A simple Web application that allows us to interrogate the database. It relies on the service façades in clinic-service to access the database.

There are two service façades, for patients and providers. The interface for patients is as follows:

```
public interface IPatientService {
	public UUID addPatient(PatientDto dto) throws PatientServiceExn;
	public List<PatientDto> getPatients() throws PatientServiceExn;
	public PatientDto getPatient(UUID id) throws PatientServiceExn;
	public TreatmentDto getTreatment(UUID patientId, UUID treatmentId) ...;
	public void removeAll() throws PatientServiceExn;
}
```

The interface for providers is as follows:

```
public interface IProviderService {
	public UUID addProvider(ProviderDto dto) throws ...;
	public List<ProviderDto> getProviders() throws ...;
	public ProviderDto getProvider(UUID id) throws ...;
	public UUID addTreatment(TreatmentDto dto) throws ...;
	public TreatmentDto getTreatment(UUID providerId, UUID treatmentId) ...;
	public void removeAll() throws ProviderServiceExn;
}
```

The implementations for these interfaces access the database by injecting DAOs and using the DAO operations. *These service façades are the only way your application should access the domain model.* The insertion operations construct entity objects from DTOs and insert them into the database. The retrieval operations query the database for entity objects and return DTOs constructed from those entity objects. The implementations of the service

facades respect the aggregate pattern of the domain model, and define exporter operations that construct DTOs for treatments from the contents of the entity objects:

```
public class TreatmentExporter implements ITreatmentExporter<TreatmentDto> {

	private boolean includeFollowups;

	@Override
	public TreatmentDto exportDrugTreatment(UUID tid,
			UUID patientId,
			String patientName,
			UUID providerId,
			String providerName,
			String diagnosis,
			...
			Supplier<Collection<TreatmentDto>> followups) {
		DrugTreatmentDto dto = factory.createDrugTreatmentDto();

		// Initialize the fields of the DTO

		if (includeFollowups) {
			dto.setFollowupTreatments(followups.get());
		}
		return dto;
	}
}
```

When exporting follow-up treatments, we must be careful not to unnecessarily query for the follow-up treatments if they are not required. The `ITreatmentExporter` interface specifies a callback for the follow-up treatments[1], that the exporter implementation can call to perform the query if necessary. The use of lambdas (closures) like this, suspending a computation until needed, is a standard way to implement lazy evaluation.

Initialization now always goes through the service layer to access the database, by injecting the service facades:

```
@Singleton
@LocalBean
@Startup
public class InitBean {

	@Inject
	private IPatientService patientService;

	@PostConstruct
	public void init() {
		PatientDto john = patientFactory.createPatientDto();
		...
		patientService.addPatient(john);
	}
```

[1] The `Supplier` interface in Java is intended for functions (lambdas) that take zero arguments and return a result.

}

The Web application uses Java Server Faces (JSF) to present the data. There are three main folders for a Maven Web application:

1. `res/main/java`: This is the folder for Java source code. There is one Java backing bean for each Web page, that stores the current state of the page (e.g., the result of querying the database) and provides operations that may be invoked as part of rendering the page.
2. `res/main/resources`: This is the folder for resources that are deployed in the jar file for the application. The main resource of note here is `Messages.properties`, a properties file that contains default text for any Web output. These defaults can be overwritten by other language-specific properties file, selected based on the locale of the JVM.
3. `res/main/webapp`: This is the Web content for the application, including images and style sheets. A Web page is described using HTML and the JSF markup language, from which an HTML page is rendered and returned to the client. A simple expression language (EL) allows reference in a page to operations and data in the backing bean.

For example, a Web page showing a list of patients uses the `dataTable` element in the JSF markup language to render a table in HTML containing patient information. The list of patients is obtained by referencing the "`patients`" property of the backing bean:

```
<p align="center">
<h:dataTable styleClass="gridTable"
             value="#{patientsBacking.patients}"
             var="patient">
    <h:column>
        <h:link outcome="view-patient">
            <f:param name="id" value="#{patient.id}" />
            <h:outputText id="name" value="#{patient.name}" />
        </h:link>
    </h:column>

    <h:column>
        <h:outputText id="dob" value="#{patient.dob}">...</h:outputText>
    </h:column>
</h:dataTable>
</p>
```

The backing bean for this Web page, named "`patientsBacking`", is defined as follows:

```
@Named("patientsBacking")
@RequestScoped
public class PatientsBacking {

       @Inject
       IPatientService patientService;

       private List<PatientDto> patients;

       public List<PatientDto> getPatients() {
```

```
            return patients;
        }

        @PostConstruct
        private void init() {
            patients = patientService.getPatients();
        }

}
```

The bean is scoped for the duration of a Web request and stores a list of patients that it obtains by calling into the patient service façade to query the database. The EL expression in the `dataTable` element retrieves the list of patients from this backing bean. The `dataTable` element then loops over each patient in this list, using the `id`, `name` and `dob` fields in each patient object (referenced by the `patient` loop variable) to render data and links in the table row for that object.

The configuration of the Web application is provided by descriptors in the `WEB-INF` folder (e.g., `faces-config.xml` that describes the navigational structure of the Web application and specifies `Messages.properties` as the resource bundle containing the definitions of the message keys for output on Web pages).

Submission

You need to complete the following projects:

1. `clinic-service`: Complete any operations that have been left unfinished, as well as providing any necessary CDI annotations for defining or injecting beans.
2. `clinic-init`: Provide additional test data, as you did for the previous assignment. Remember that you should only access the database using the patient and service facades. The DAOs should **not** be injected into the initialization bean.
3. `clinic-webapp`: Complete any CDI annotations that are required for the backing beans, to enable your patient and provider facades to be injected into the presentation layer where needed. Remember that you should only access the database using the patient and service facades. The DAOs should **not** be injected into the backing beans.

You are given the clinic-root project, which you should use to compile your projects and build your application as a WAR file, `clinic-webapp.war`. You should reuse your `clinic-domain` and `clinic-dto` projects from previous assignments. Any changes that need to be made for this assignment will be explained. As with the previous assignment, you should deploy your application in the Payara server (locally or on EC2, whichever you prefer, for now), using Postgresql as the backend database. You still have no way of inputting data after the initialization in clinic-init, but at least you can interrogate the database using the Web application. So, it is important that you provide sufficient test data during initialization to demonstrate your service logic working.

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 your updates of the Maven projects provided for this assignment. You should also provide:

1. a rubric that records what you accomplished, and
2. videos demonstrating the working of your assignment.

Finally, the root folder should contain the Web archive file (`clinic-webapp.war`) that you used to deploy your application.

Your solution should be uploaded via the Canvas classroom, as a zip file. This zip file should have the same name as your Canvas/Stevens username. It should unzip to a folder with this same name, which should contain the files and subfolders with your submission.

It is important that you provide a completed rubric that documents your submission, included as a PDF document in your submission root folder. As part of your submission, export all the Maven projects for this application to your file system, and then include those folders as part of your zip file.