# CS 548—Fall 2023
# Enterprise Software Architecture and Design
# Assignment Four—Domain Driven Design

Provide a domain-driven design for a clinical information system. You will use the data model you designed in the previous assignment. You should add domain-specific logic to the Java classes of the last assignment. Follow the principles of domain-driven architecture in designing your domain model. You are provided with several Maven projects that you should complete for the assignment:

- `clinic-webapp0`: A dummy Web app for the project.
- `clinic-domain`: The domain model for the app.
- `clinic-init0`: Initializes the app during deployment.
- `clinic-root`: A Maven parent project for the modules above.

You should open these projects in Intellij IDEA Ultimate by opening the POM file of `clinic-root` (You may need to reload the Maven projects). Use the Intellij Maven tool to compile the projects and generate the WAR file, `clinic-webapp.war`, for the Web app deployed from this project. We are using Maven to manage dependencies, between projects and with external software. The Web app currently does not do anything except execute the initialization logic in `clinic-init0`, which in turn executes the logic in the domain model.

The persistence descriptor for the domain project, `persistence.xml` in `clinic-domain`, in the `META-INF` folder under resources, is configured to create or extend the database tables each time the app is deployed. You will need to edit it to list the entity classes.

## Factory and Repository (DAO) Pattern

Use the *factory pattern*. Define factory objects for creating patient, treatment and provider entities. Define *repository (DAO) objects* that encapsulate the use of the entity manager. The entity manager should **not** be accessed outside a DAO. There should be one DAO per entity type that is persisted (patient, provider and treatment). These DAOs should be defined as CDI beans, which are then injected where required in managed beans (See the initialization logic in `clinic-init0`):

```
@RequestScoped
@Transactional
public class PatientDao implements IPatientDao {


   @Inject @ClinicDomain
   private EntityManager em;
```

The producer of the entity manager is also defined as a CDI bean:

```
@RequestScoped
@Transactional
public class ClinicDomainProducer {

    @PersistenceContext(unitName="clinic-domain")
    EntityManager em;
    @Produces @ClinicDomain
    public EntityManager clinicDomainProducer() {
       return em;
    }
```

This is the **only** place where you should use the `PersistenceContext` annotation.

Note that a follow-up treatment is also between a patient and provider (though not necessarily the same provider as for the original treatment).  Make sure to include the logic for programmatically maintaining the relationships that exist between patients, providers and treatments, as entities are inserted. If you do not do this, you will get an exception when your program tries to flush your object graph to the database, due to violation of database integrity constraints.

A patient or provider entity will need a treatment DAO to access and create treatment entities. How do they obtain such an object, since like all JPA objects patients and providers are POJOs rather than managed beans? One pattern to achieve this is to add a transient field for the DAO in these entity objects, and then provide a setter method to set the DAO in the entity when it is persisted or retrieved. For example:

```
@Table(indexes = @Index(columnList="patientId"))
public class Patient implements Serializable {

   @Transient
   private ITreatmentDao treatmentDao;

   public void setTreatmentDao (ITreatmentDao tdao) {
      this.treatmentDao = tdao;
   }

}
```

## Aggregate and Visitor Pattern

Architect patient and provider *aggregate objects* for accessing treatment information. These aggregates encapsulate the underlying treatment entity objects and provide the logic for accessing treatment information without returning treatment objects. To support this, a treatment entity supports the *visitor pattern*: A "visit" operation that takes a client callback object with four methods, one for each of the forms of treatment:

```
public interface ITreatmentExporter<T> {

   public T exportDrugTreatment(UUID tid, UUID patientId, String patientName, ...);

   public T exportRadiology(UUID tid, UUID patientId, String patientName, ...);

   public T exportSurgery(UUID tid, UUID patientId, String patientName, ...);

   public T exportPhysiotherapy(UUID tid, UUID patientId, String patientName, ...);

}
```

A treatment entity uses a visitor object to expose its state to its client without exporting the treatment entity itself, e.g., in DrugTreatment:

```
public <T> T export(ITreatmentExporter<T> visitor) {
  return visitor.exportDrugTreatment(treatmentId,
                                     patient.getPatientId(),
                                     patient.getName(),
                                     ...
                                     () -> exportFollowupTreatments(visitor));
}
```

We will use this pattern in the next assignment for extracting information from the domain model without violating the aggregate pattern. For now, the initialization bean in this assignment implements `ITreatmentExporter<Void>`; it does not extract data but is used to log the contents of the database:

```
Patient john = patientFactory.createPatient();
john.exportTreatments(this);

@Override
public Void exportDrugTreatment(... String name, ... String drug, ...,
                                Supplier<Collection<Void>> followups) {
   logger.info(String.format("Drug treatment for %s, drug %s", name, drug));
   followups.get();
   return null;
}
```

There is one subtlety in the use of the visitor pattern for exporting treatment information: We may not necessarily want to export the follow-up treatments for a treatment. So, an exporter operation has the signature e.g.,

```
public T exportDrugTreatment(..., Supplier<Collection<T>> followups);
```

Instead of eagerly exporting the follow-up treatments, an exporter operation provides a *supplier*, a function of zero arguments that performs the export of the follow-up treatments if they are needed. For example, the exporter for drug treatments above provides a lambda that will export the follow-up treatments if invoked:

```
() -> exportFollowupTreatments(visitor)
```

The consumer of the exported treatment calls the `get()` method on the supplier if the follow-up treatments are needed. This is shown in the example above, where the follow-up treatments are displayed after the drug treatment in the logs when the supplier is executed (`followups.get()`).

The export operations allow treatment information to be exported outside patient and provider aggregates without allowing treatment entities to be “leaked” out of the aggregates, but we also must be able to import treatment information into an aggregate, where the entity object is created and persisted to the database. To support this, a provider entity has import operations, one for each of the possible treatments, e.g.,

```java
@Override
public Consumer<Treatment> importDrugTreatment(UUID tid, Patient patient, ...
                                              Consumer<Treatment> consumer)
{
   DrugTreatment treatment = treatmentFactory.createDrugTreatment();
   ...
   provider.addTreatment(treatment);
   patient.addTreatment(treatment);
   treatmentDao.addTreatment(treatment);
   ...
}
```

There is a private operation for adding a treatment to the list of treatments for a provider:

```java
private void addTreatment (Treatment t) {
   treatments.add(t);
   t.setProvider(this);
}
```

There is a package-visible operation for adding a treatment to the list of treatments for a patient:

```java
void addTreatment (Treatment t) {
   treatments.add(t);
   t.setPatient(this);
}
```

Both of these operations are called by one of four public import treatment operations defined in the provider[1], one for each type of treatment.

The challenge again is with follow-up treatments: We cannot create a list of follow-up treatments to pass to the aggregate, because treatments can only be created inside the aggregate. We will follow this protocol: When we create a new treatment entity, we return not the entity object but a *consumer* function, a function that accepts a follow-up treatment entity as its single argument and produces no result. Our consumer adds any treatment entity it is provided with to the list of follow-up treatments for the treatment for which it was created. So, the last line of `importDrugTreatment` above is:

```java
   return (followUp) -> { treatment.addFollowupTreatment(followUp); };
```

As we then import the follow-up treatments for this treatment, we supply an extra argument to the import operation, a consumer function that adds each treatment entity that is generated to a list of follow-up treatments. At the top-level, where a treatment is being added that is not a follow-up treatment, this consumer is null, otherwise the import operation calls the consumer to add the treatment to the consumer treatment's follow-up treatments.:

[1] This is why the `addTreatment` operation in the `Patient` class has package visibility.

```
   if (consumer != null) {
      consumer.accept(treatment);
   }
```

Here is an example of a top-level importation of a treatment:

```
jane.importDrugTreatment(UUID.randomUUID(), john, ..., null);
```

## Testing

Since the application does not yet have an external interface, do your testing in the initialization bean, displaying the results of testing in the log in the application server. Testing your code now will avoid postponing all your testing until a later assignment, when you deploy with an external interface. A sketch of the definition of the initialization bean is provided to you below. You should fill this in with your own initialization logic in the `init()` method.

```
@Singleton
@LocalBean
@Startup
public class InitBean implements ITreatmentExporter<Void> {

   @Inject
   private IPatientDao patientDao;

   @Inject
   private IProviderDao providerDao;

   @PostConstruct
   public void init() {
      Logger.info("Your name here: ");
      ...
   }
```

This bean injects patient and provider DAOs, to persist entities of these types. Treatment entities are persisted and retrieved internally in the patient and provider aggregates , using their internal treatment DAOs.

We specify the database connection for this application with this annotation on the declaration of the `AppConfig` class in the web application module `clinic-webapp0`:

```
@DataSourceDefinition(
       name="java:global/jdbc/cs548",
       className="org.postgresql.ds.PGSimpleDataSource",
       user="${ENV=DATABASE_USERNAME}",
       password="${ENV=DATABASE_PASSWORD}",
       databaseName="${ENV=DATABASE}",
       serverName="${ENV=DATABASE_HOST}",
       portNumber=5432)
```

This specifies a database connection pool that is created when the app is started. This connection pool is identified by the global JNDI name `java:global/jdbc/cs548`. The persistence descriptor in `clinic-domain` specifies the same global JNDI name for the database connection, connecting this database connection pool to your application.

The data source definition also specifies the credentials for accessing the database, the database name, and the host and port for the database server. The database name and host, and database username and password, are taken from environment variables (using `ENV`) when the application is deployed, one of the best practices suggested for 12-Factor applications. The dependencies in the web application include the Jakarta EE API, the initialization bean and the JDBC driver for the Postgresql database (with the versions specified in the root module POM file):

```
<dependency>
   <groupId>jakarta.platform</groupId>
   <artifactId>jakarta.jakartaee-api</artifactId>
</dependency>
<dependency>
   <groupId>org.postgresql</groupId>
   <artifactId>postgresql</artifactId>
</dependency>
<dependency>
   <groupId>edu.stevens.cs548</groupId>
   <artifactId>clinic-init0</artifactId>
</dependency>
```

You have two options for deploying an app with this domain model. 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-payara`, move the war file there, and navigate into this directory:

```
$ mkdir cs548-payara
$ mv clinic-webapp.war cs548-payara
$ cd cs548-payara
```

In this directory, create a file called `Dockerfile` that copies the WAR file for the application to the container file system[2]:

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

Save this file, and create a custom server image:

```
$ docker build -t cs548/clinic .
```

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

```
$ 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 --network cs548-network -p 8080:8080 -p
8181:8181 -e DATABASE_USERNAME=cs548user -e DATABASE_PASSWORD=YYYYYY -e
DATABASE=cs548 -e DATABASE_HOST=cs548db cs548/clinic
```

The web application is a dummy application for this assignment. For now, the only way to see output is to see what has been written by the initialization code in the logs, using `docker logs`.

## 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[3], 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 app), run the application as follows (on MacOS and Linux)[4]:

```
$ 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-webapp.war --contextroot clinic
```

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-webapp.war --contextroot clinic
```

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.

[3] https://www.payara.fish/downloads/payara-platform-community-edition/
[4] You may also want to redirect output to a file, so you can examine it with an editor. See the first assignment spec.

Remember that you will need to delete and recreate the database whenever the schema changes:

```
$ docker run -it --rm --network cs548-network postgres psql -h cs548db -U
postgres
postgres=# drop database cs548 with (force);
postgres=# create database cs548 with owner cs548user;
postgres=# \q
```

Whichever way you deploy the web app, since the web app only has a dummy user interface, for now it is only able to add data to the database in the initialization code in `InitBean`. Use Intellij's database tool window to connect to the database server and show the successful addition of entity records to the database after the app has been deployed.

## Submission

**Your solutions should be developed for Jakarta EE 10. You should deploy your WAR file to Payara Server version 6.2023.9. You should use Java 17 with Intellij IDEA Ultimate.** In addition, record short mpeg videos of a demonstration of your assignment working (deploy the app and view the server logs in enough detail to see the output of your testing). 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.* You can upload this video to the Canvas classroom.

Your solution should consist of a zip archive with one folder, identified by your name. Within that folder, you should have four Intellij Maven projects: `clinic-webapp0`, `clinic-root`, `clinic-domain` and `clinic-init0`. You should also provide videos demonstrating the working of your assignment. This involves showing your project being successfully deployed and viewing the logs to see the output of successful testing. Finally, the root folder should contain the WAR file (`clinic-webapp.war`) that you used to deploy your application. You should also provide a completed rubric.