CS 548—Fall 2023
Enterprise Software Architecture and Design
Assignment Three—Object Relational Mapping

Provide the data model for a domain-driven design for a clinical information system. You will define a data model using JPA-annotated Java code. You will complete a command line program that adds entities to the database, and you can confirm the addition of the entities using the Intellij database tool window. In the next assignment, you will add domain-specific logic to the Java classes.

Data Model

The clinic data model consists of a collection of patient entities, one for each patient. Each patient entity has:

1. a patient identifier, a universally unique identifier (UUID), automatically generated when a patient record is added to the system;
2. a patient name; and
3. a patient date of birth.

The model also consists of healthcare provider entities. Every provider entity includes their automatically generated identifier, provider identifier (NPI) which may be assigned by the government, and their name.

There is obviously a many-to-many relationship between patients and providers, but we will represent it by two one-to-many relationships: from patients to treatments received (represented by a collection of treatment entities associated with a patient entity), and from providers to treatments administered (again represented by a collection of treatment entities). Each treatment entity links back to the corresponding patient and provider. You are responsible for maintaining the object graph in memory; the ORM only ensures that the object graph is saved to a database on disk.

Every treatment entity includes, besides its own key and references to the corresponding patient and provider, a diagnosis of the condition for which the treatment is prescribed (e.g. throat cancer, HIV/AIDS, hepatitis, etc). The remaining information for a treatment depends on what form of treatment it is. Currently there are four specific forms of treatment records:

1. A drug treatment record includes a drug and a dosage, starting and ending date for the treatment, and frequency (number of times a week, an integer).
2. A surgery treatment record includes a date of surgery, discharge instructions and a collection of follow-up treatments (e.g., drug treatment and physiotherapy).
3. A radiology treatment record includes a list of dates of radiology treatments, and a collection of follow-up treatments.
4. A physiology treatment record includes a list of dates of physiotherapy treatments.

Each entity type (patient, provider or treatment) includes two keys, an internal database key (a long integer) generated by the database server and an external UUID key generated by the application. For example, here is part of the patient entity class (with some annotations missing):

```
public class Patient implements Serializable {
```

```
    private long id;

    private UUID patientId;

}
```

The id field is the internal database key, while the UUID field is the external application key. Applications search for entities based on their UUID key. The entity class should be annotated with a `@Table` annotation that specifies a secondary index on this key to make this a fast search:

```
@Table(indexes = @Index(columnList="patientId"))
```

This column should also be non-nullable and unique, since it is intended to be a secondary key:

```
@Column(nullable = false, unique = true)
```

You should specify the relationships between entities with `@OneToMany` and `@ManyToOne` annotations, specifying that persistence propagates *in both directions*, e.g.,

```
public class Patient implements Serializable {

    @OneToMany(cascade = PERSIST, mappedBy = "patient")
    private Collection<Treatment> treatments;

}
```

The exception to this is when the related objects are simple values rather than entities, e.g., the list of dates for physiotherapy. You can specify this using the `@ElementCollection` annotation:

```
@ElementCollection
@OrderBy
protected List<LocalDate> treatmentDates;
```

Since the reference is to a list of dates, you must define an order on them. The `@OrderBy` annotation specifies the default ordering for dates.

Remember that entity objects are not managed, you are responsible for setting their data. The ORM is only responsible for persisting entities (and updates to those entities) to a database and retrieving entities (including related entities) from a database. For example, you must initialize any lists of related entities in the constructor:

```
public Patient() {
   super();
   treatments = new ArrayList<Treatment>();
}
```

When a treatment is added, you have to set the forward and backward links from the patient entity (adding the treatment to that patient's list of treatments) and setting the

treatment's patient link. Similarly for the provider administering that treatment. See `Patient::addTreatment` and `Provider::addTreatment`. The logic for adding each treatment sets these forward and backward links in the object graph:

```
msg("Patient ID: ");
UUID patientId = UUID.fromString(in.readLine());
Patient patient = getPatient(patientId);
patient.addTreatment(treatment);

msg("Provider ID: ");
UUID providerId = UUID.fromString(in.readLine());
Provider provider = getProvider(providerId);
provider.addTreatment(treatment);
```

Any database updates have to be performed in the context of a transaction, that should be rolled back if a fault happens:

```
public void addTreatmentToDatabase() throws IOException, ParseException {
   entityManager.getTransaction().begin();
   try {
      Treatment treatment = addTreatment();
      if (treatment != null) {
         /*
          * Save the record in the database.
          */
         entityManager.persist(treatment);
         entityManager.getTransaction().commit();
      }
   } catch (Exception e) {
      entityManager.getTransaction().rollback();
      throw e;
   }
}
```

In addition to completing the entity classes, you must also list these classes in the persistence descriptor, `META-INF/persistence.xml`.

Test Application

You are provided with a command line program for adding records to a database[1]. The Maven dependencies for this project include the Jakarta Persistence API, the EclipseLink implementation of this API, and the JDBC driver that is specific to the database management system we are connecting to (PostgreSQL):

```
<dependency>
   <groupId>org.eclipse.persistence</groupId>
   <artifactId>org.eclipse.persistence.jpa</artifactId>
   <version>...</version>
</dependency>
<dependency>
   <groupId>jakarta.persistence</groupId>
```

[1] You will need to run the Postgresql database server in a Docker container, either in EC2 or locally using Docker Desktop, and update the database URL in `client.properties` accordingly.

```
   <artifactId>jakarta.persistence-api</artifactId>
   <version>...</version>
</dependency>
<dependency>
   <groupId>org.postgresql</groupId>
   <artifactId>postgresql</artifactId>
   <version>...</version>
</dependency>
```

You need to complete the code for adding treatment entities, similar to the previous assignment. This program connects to a database using properties defined in file `client.properties` defined as a resource in the project. You can override these properties on the command line, and you should provide the password for the database user on the command line:

```
$ java -jar clinicdb.jar --password <database-password>
```

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 the web app assumes the database server is running on the same machine as the web app, specifying `localhost` as the database host in the JDBC URL for the database server (see the file `client.properties` in the resources folder). If the database server is running in EC2, you will need to specify a JDBC URL that defines the external IP address of the EC2 instance as the target host:

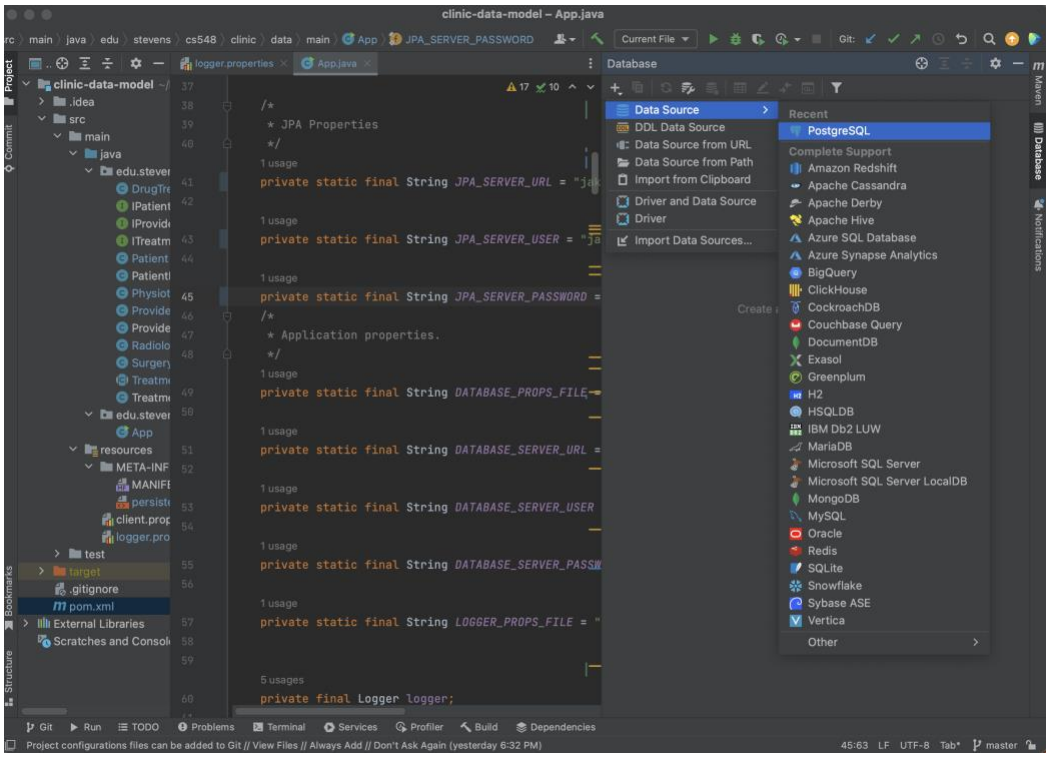
```
$ java -jar clinicdb.jar --password <database-password> --server <jdbc-url>
```

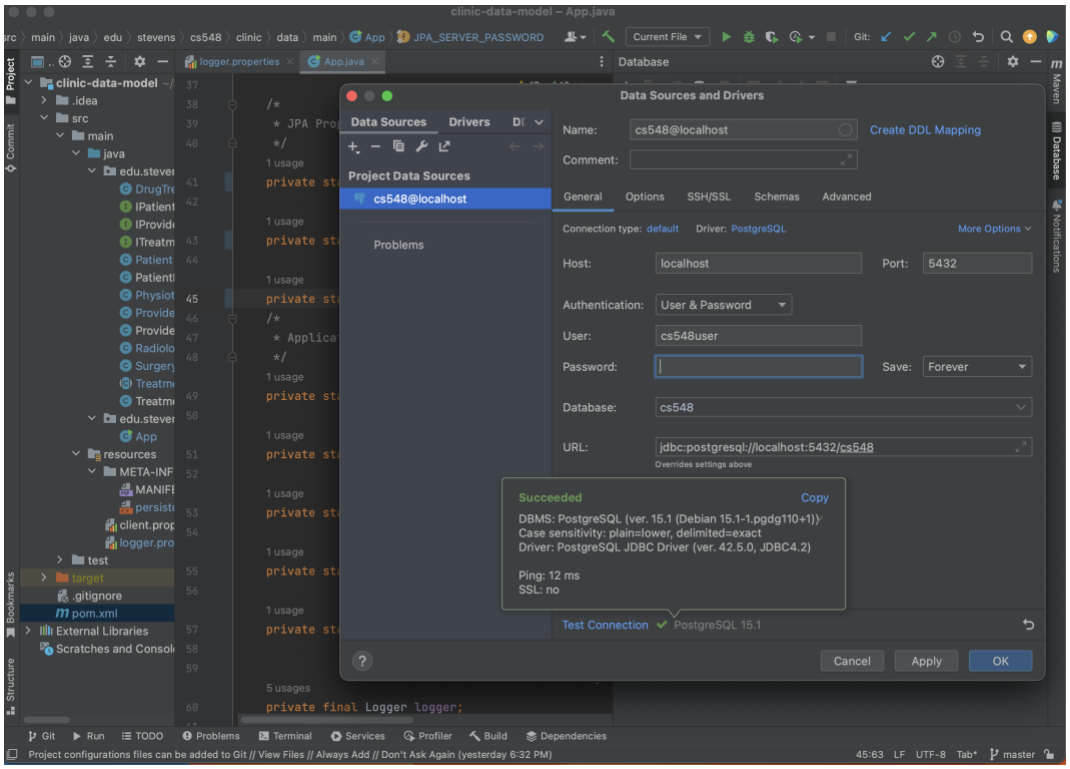
In either case, you will need to delete and recreate the database whenever the schema changes:

```
$ 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
```

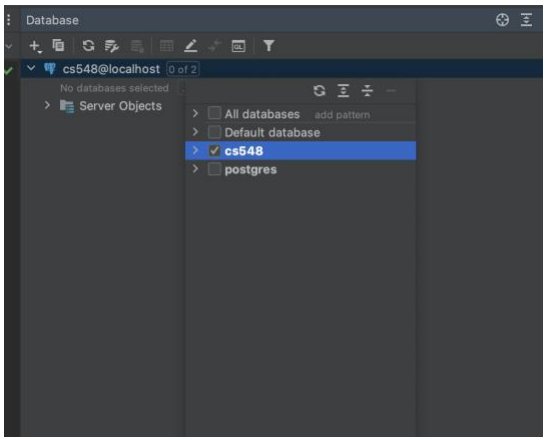
You can view the SQL commands including DDL sent to the database logged in the server log file (default `database.log`, specified in the `logger.properties` file). You should submit this log file with your submission. You should also provide a MPEG video showing the running of this application adding patients, providers and treatments (at least one of each type of treatment), and verifying that the data has been added to the database. For this last confirmation, you should open the Database tool in Intellij[2], create a connection to the database server, and view the data in the database tables:

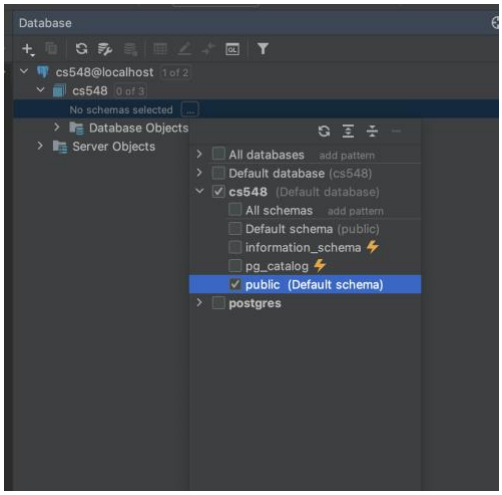
[2] Go to `View | Tool Windows | Database`.



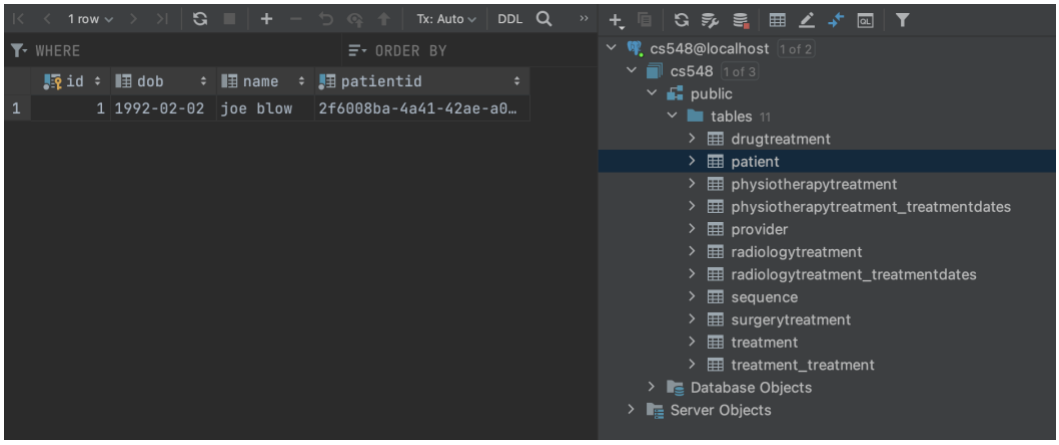


Once you are connected to the database server, you need to select the database and the public (default) schema:



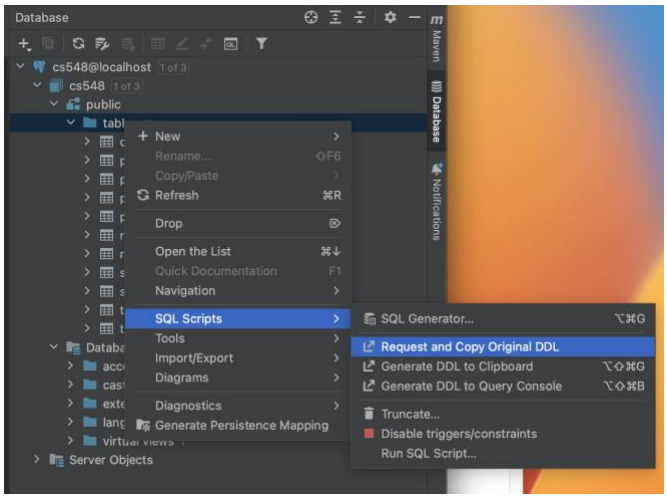


Now you will be able to view the schema for the database and sample the data by double-clicking on a table[3]:



You can also obtain the data definition language (DDL) script for the database or individual tables:

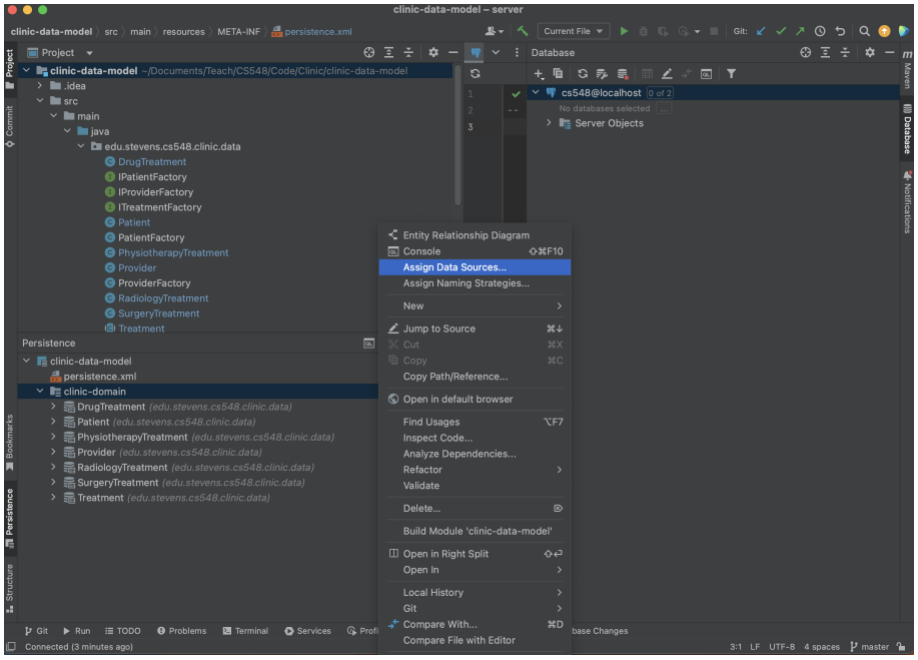
[3] https://www.jetbrains.com/help/idea/tables-view-data.html.



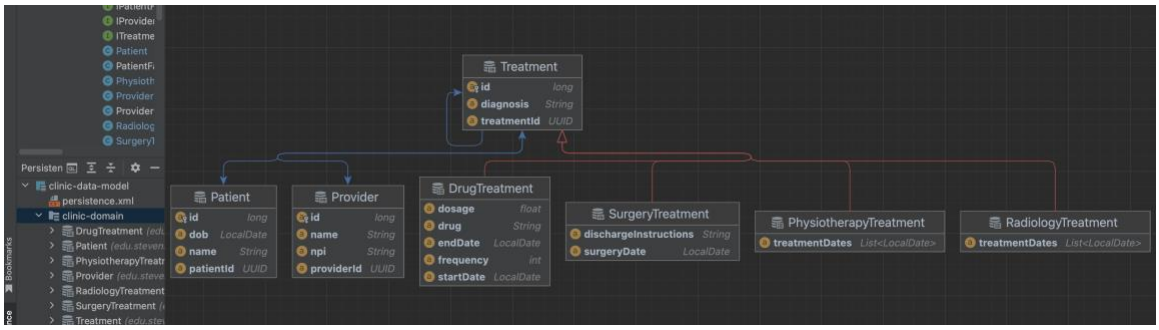
For example, here is the DDL for the `Patient` table:

```
create table public.patient (
    id        bigint       not null primary key,
    dob       date,
    name      varchar(255),
    patientid varchar(255) not null unique
);

alter table public.patient owner to cs548user;

create index index_patient_patientid
    on public.patient (patientid);
```

You can assign the data source to the persistence unit for the application using the Persistence tool (`View | Tool Windows | Persistence`):



Using this for example you can view an entity-relationship diagram for your data model (Note the one-to-many relationship from `Treatment` to itself because of follow-up treatments):



Make sure that your name appears at the beginning of the video. For example, display the contents of a file that provides your name. *Do not provide private information such as your email or cwid in the video*. Be careful of any "free" apps that you download to do the recording, there are known cases of such apps containing Trojan horses, including key loggers.

Submission

Your solution should be uploaded via the Canvas classroom, as a zip file. This zip file should have the same name as your Canvas userid. It should unzip to a folder with this same name, which should contain the files and subfolders with your submission (including your completed Intellij project).

As part of your submission, do a Maven clean of your Intellij project, and then include that project in a folder as part of your archive file. You should also provide a video demonstrating running the (completed) test application to populate the database with data and viewing the results in the Intellij Tool Window. You should also provide the DDL for the database and your client log file database.log from this session. Finally, you should provide a completed rubric.