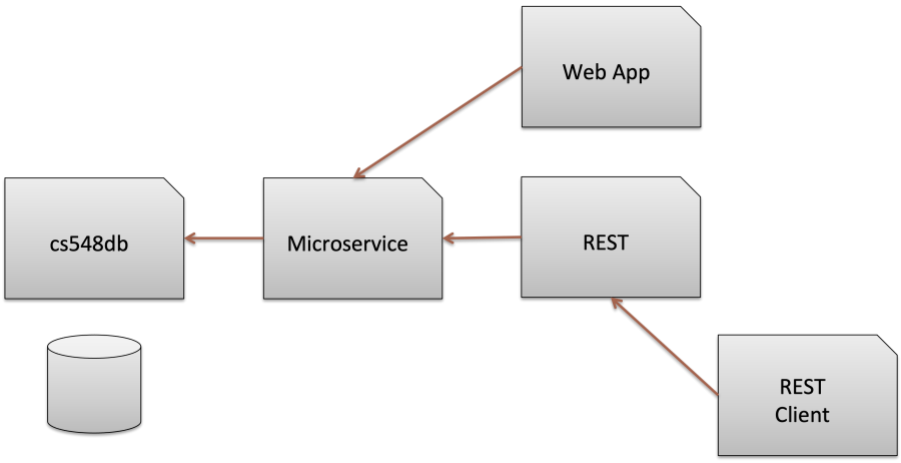
CS 548—Fall 2023
Enterprise Software Architecture and Design
Assignment Eight—Micro Profile

In this assignment, you will deploy the applications from previous assignments as microservices, using Payara Micro. The architecture is as follows:



There are two new Maven projects for this assignment:

1. `clinic-microservice` is the implementation of a microservice that "wraps" access to the database, and all frontend apps that want to access the database must go through this. The domain model is only accessed in this project. Maven generates a WAR file for this microservice called `clinic-domain.war`.
2. `clinic-microservice-client` is a library that should be linked into the frontend apps, encapsulating use of the microservice and implementing the interfaces in `clinic-service-client`. The `clinic-service` project is no longer used for this assignment since this service layer is now replaced by the backend microservice and the microservice client library that provides access to it.

Step 1: Deploy the database server

First, create a shell script `cs548db.sh` that will be executed in the database container after it has initialized, to create the user and database for the application:

```
#!/bin/bash
set -e
psql -v ON_ERROR_STOP=1 \
     -v password="$DATABASE_PASSWORD" \
     --username "$POSTGRES_USER" \
     --dbname "$POSTGRES_DB" <<-EOSQL
    CREATE USER cs548user PASSWORD :'password';
    CREATE DATABASE cs548 WITH OWNER cs548user;
    GRANT ALL PRIVILEGES ON DATABASE cs548 TO cs548user;
EOSQL
```

Make sure that file permissions allow any user to read and execute this shell script. The Postsgresql docker script expects the superuser password (default user `postgres`) to be specified in the environment variable `POSTGRES_PASSWORD`. This script in addition expects the application password to be specified in the environment variable `DATABASE_PASSWORD`, which is then passed to the SQL script using the SQL variable `password`.

Next, create a dockerfile that will initialize the default superuser with a password you specify (see below), and copies this shell script to a location where Postgresql will execute it once the server has been initialized:

```
FROM postgres
COPY --chown=postgres:postgres cs548db.sh /docker-entrypoint-initdb.d/
```

This shell script will now be executed when the database server is first started. Create the image in your local docker repository:

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

You can now run this database server as you did for the first assignment, for example:

```
$ docker run -d --name cs548db --network cs548-network -p 5432:5432 -v path-
to-host-data-dir:/var/lib/postgresql/data -e POSTGRES_PASSWORD=XXXXXX -e
PGDATA=/var/lib/postgresql/data/pgdata -e DATABASE_PASSWORD=YYYYYY
cs548/clinic-database
```

Microservices will communicate with the database at port 5432, on "virtual host" `cs548db` on the virtual network you will have set up. They will authenticate as user `cs548user`, with the database password specified here at the time that the service is started. The database superuser password is provided in order to enable these changes to be made. The port is exposed here on the host machine to allow you to connect to it e.g., from Intellij, using the Database tool window.

Step 2: Complete and deploy the domain microservice

Access to the database will be restricted to a single microservice, implemented by a Web service in a file `clinic-domain.war` generated from the `clinic-microservice` project. This microservice exposes two REST resources, for patients and providers. It provides exactly the same functionality as the `clinic-service` project from previous assignments, except now exposing those services as Web services[1]. For example, here is the operation for creating a patient; it returns the identifier for the newly added patient record as a uniform resource identifier (URI) in the `Location` HTTP response header, with status code 201 (`Created`):

```
@POST
@Consumes("application/json")
public Response addPatient(PatientDto dto) {
   try {
```

[1] Remember to declare the resource classes for the microservice as transactional, since they are accessing the domain model directly.

```
            Patient patient = patientFactory.createPatient();
            ...
            patientDao.addPatient(patient);
            UUID id = patient.getPatientId();
            URI uri = uriInfo.getBaseUriBuilder().path(id.toString()).build();
            return Response.created(uri).build();
        } catch (PatientExn e) {
            logger.log(Level.SEVERE, "Failed to add patient!", e);
            return Response.serverError().build();
        }
    }
```

The operation for returning a list of all patients is an example of returning an HTTP error status code by throwing `WebApplicationException`:

```
    @GET
    @Produces("application/json")
    public List<PatientDto> getPatients() {
        try {
            Collection<Patient> patients = patientDao.getPatients();
            List<PatientDto> dtos = new ArrayList<>();
            for (Patient p : patients) {
                dtos.add(patientToDto(p, false));
            }
            return dtos;
        } catch (Exception e) {
            logger.log(Level.SEVERE, "Failed to get patients!", e);
            throw new WebApplicationException(Status.INTERNAL_SERVER_ERROR);
        }
    }
```

The microservice also includes health and readiness checks. The health check ensures that the JVM has sufficient memory (based on a threshold specified as a configuration property):

```
    @Liveness
    @ApplicationScoped
    public class LivenessCheck implements HealthCheck {

        @Inject
        @ConfigProperty(name = MEMORY_THRESHOLD_PROPERTY)
        private long threshold;

        @Override
        public HealthCheckResponse call() {
            HealthCheckResponseBuilder responseBuilder =
                        HealthCheckResponse.named(LIVENESS_CHECK_NAME);
            ...
            return responseBuilder.up().build();
        }
    }
```

The readiness check pings the database server to make sure it is still available:

```
Readiness
@ApplicationScoped
public class ReadinessCheck implements HealthCheck {

   @Override
   public HealthCheckResponse call() {
      HealthCheckResponseBuilder responseBuilder =
            HealthCheckResponse.named(READINESS_CHECK_NAME);
      ...
      return responseBuilder.up().build();
   }
}
```

These health checks can be queried at these context paths once the microservice is deployed:

```
/health/live
/health/ready
/health
```

As with the previous assignments, you have two options for deploying the microservice.

Containerized Deployment Using Payara Micro

On your laptop or on EC2[2], create a directory called `cs548-domain`, move the `clinic-domain.war` file there, and navigate into this directory:

Once you have completed the definitions of the methods in the `PatientMicroService` and `ProviderMicroService` classes, in the same way you did for `PatientService` and `ProviderService` in an earlier assignment, and the health checks, you should build the image for the microservice. Create this dockerfile for the microservice:

```
FROM payara/micro:6.2023.2-jdk17

COPY --chown=payara:payara clinic-domain.war ${DEPLOY_DIR}

CMD [ "--contextroot", "api", \
      "--deploy", "/opt/payara/deployments/clinic-domain.war" ]
```

This dockerfile copies the WAR file for the application to the container file system. This file should be in the same directory as the dockerfile. The JDBC driver is specified as a Maven dependency and copied into the WAR file by Maven. Then build the docker image:

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

You run this microservice as a container in a similar way to earlier assignments, e.g.:

```
$ docker run -d --name clinic-domain --network cs548-network -p 5050:8080 -e
DATABASE_USERNAME=cs548user -e DATABASE_PASSWORD=YYYYYY -e DATABASE=cs548 -e
DATABASE_HOST=cs548db cs548/clinic-domain
```

[2] There may be an issue with health checks working on EC2, in which case you will have to demonstrate the microservice health checks working on your laptop.

You specify the properties for the database connection as environment variables when you run the microservice[3]. Note that in general you should not expose the ports for this microservice, since it should only be accessed as a backend by the frontend services. We are exposing the microservice outside the virtual network at port number 5050 to demonstrate the health and readiness checks.

Native Deployment Using Payara Micro

You can deploy the application natively (for now) on your laptop. Download the payara micro jar file[4], say to a file called `payara-micro.jar`, and then (in a directory that contains the jar files for both Payara Micro and the microservice app), run the application as follows (on MacOS and Linux)[5]:

```
$ export DATABASE_USERNAME="cs548user"
$ export DATABASE_PASSWORD="YYYYYY"
$ export DATABASE="cs548"
$ export DATABASE_HOST="localhost"
$ java [ -Djava.net.preferIPv4Stack=true ]
    --add-opens=java.base/java.io=ALL-UNNAMED -jar payara-micro.jar
    --deploy clinic-domain.war --contextroot api --port 5050
```

Step 3: Complete the microservice client and run the frontend applications

The code for the frontend Web application `clinic-webapp` and REST service `clinic-rest` is unchanged. This is the power of dependency injection: All you need do for this assignment is to change the implementation for `IPatientService` and `IProviderService`, and dependency injection will inject this alternative implementation. The Web service provides the same functionality as the original service but exposes details of the fact that it is a Web service. So, we define a wrapper service `clinic-microservice-client` that implements `IPatientService` and `IProviderService`, by calling out to the backend clinic-domain microservice.

The service implementations in this class are client stubs that check the HTTP responses from Web service calls and throw exceptions if there are any issues with the service. They rely on the Micro Profile REST Client API to inject the actual HTTP client:

```
@RequestScoped
public class PatientService implements IPatientService {

   private static final String LOCATION = "Location";

   @Inject
```

---

[3] This is **not** a best practice. For example, anyone on the machine you are running on that looks at the currently running processes will see the database user password. It is at least better than putting the password in the source code. A better approach would be to use something like Docker Secrets to store the database credentials in an encrypted volume that is mounted by a container for this microservice.

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

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

```
    @RestClient
    IPatientMicroService patientMicroService;

    @Override
    public UUID addPatient(PatientDto dto) throws PatientServiceExn {
       try {
          Response response = patientMicroService.addPatient(dto);
          String location = response.getHeaderString(LOCATION);
          String[] uriSegments = URI.create(location).getPath().split("/");
          return UUID.fromString(uriSegments[uriSegments.length-1]);
       } catch (WebApplicationException e) {
          throw new PatientServiceExn("Web service failure.", e);
       }
    }

    @Override
    public List<PatientDto> getPatients() throws PatientServiceExn {
       try {
          return patientMicroService.getPatients();
       } catch (WebApplicationException e) {
          throw new PatientServiceExn("Web service failure.", e);
       }
    }

}
```

`IPatientMicroService` is an interface for the RESTful API that `PatientMicroService` provides, and dependency injection creates a client stub that implements this interface:

```
@RegisterRestClient(configKey="clinic-domain.api")
@RegisterProvider(GsonProvider.class)
@Path("patient")
public interface IPatientMicroService {

    @POST
    @Consumes("application/json")
    public Response addPatient(PatientDto dto);

    @GET
    @Produces("application/json")
    public List<PatientDto> getPatients();

    ...
}
```

The destination for the Web service calls is specified in `src/main/resources/microprofile-config.properties`, from which it is read by Micro Profile Config as part of creating the REST client stubs during dependency injection (where the `configKey` attribute identifies the property that specifies the URI for the service):

```
clinic-domain.api/mp-rest/uri=http://clinic-domain:8080/api/
```

The client class also registers a message body reader and writer provider, using Gson to serialize and deserialize JSON. Complete the annotations on these classes and make the following changes to the Maven POM files for `clinic-webapp` and `clinic-rest`. Replace:

```
<dependency>
    <groupId>edu.stevens.cs548</groupId>
    <artifactId>clinic-service</artifactId>
</dependency>
```

with:

```
<dependency>
    <groupId>edu.stevens.cs548</groupId>
    <artifactId>clinic-microservice-client</artifactId>
</dependency>
```

Note that you should **not** remove the dependency on `clinic-service-client`, that defines the interfaces that both apps depend on. You are replacing one implementation for these interfaces, that accesses the domain model directly, with another implementation that accesses the domain model indirectly through a microservice.

You can now build and run the Web application and Web service as in Assignment 6. They will each use the microservice client to issues requests to the microservice running in the background, encapsulating access to the database.

Submission

For your submission, you should provide all your Intellij modules for this project, and your dockerfiles (and their directories with their inputs) where you use docker containers. You should also provide videos that demonstrate:

1. Launching your services (creating the images if you are using Docker) and showing the endpoint information for these services. You should also demonstrate working health and readiness checks for the microservice.
2. Demonstrate your assignment working, as in previous assignments: Show the Web application running, and show it being updated as you run the client for the frontend Web service for uploading data.
3. Show using logs) operations being invoked in the microservice clients and being executed in the microservice itself. You should make the logging level in the domain descriptor (`persistence.xml`) coarse-grained (say `INFO` or `WARNING`) so that the log for the microservice is not flooded with EclipseLink logging information.

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. **Your video must be MP4 format!**

Your submission 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 the Maven modules required for this assignment, not just the ones modified for this assignment. The list of projects is:

1. `clinic-domain`
2. `clinic-dto`
3. `clinic-init`
4. `clinic-microservice`
5. `clinic-microservice-client`
6. `clinic-rest`
7. `clinic-rest-client`
8. `clinic-root`
9. `clinic-service-client`
10. `clinic-webapp`

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 archive files (`clinic-webapp.war`, `clinic-rest.war`, `clinic-domain.war`, and `clinic-rest-client.jar`) that you used to demonstrate your running application.

It is important that you provide your source code modified for this assignment, as well as dockerfiles and their inputs if you use Docker. You should also provide a video demonstrating setting up and running your services. It is sufficient to demonstrate the apps running on your own machine. You should also provide a completed rubric.