# NiQ Exercise Guide - Neil Jetty

## Set up the database – Oracle XE 18.4

1. I decided to create the data model first as I wanted to give jOOQ a try as I understand you guys use it and it works well against pre-existing databases, reverse engineering the data model/meta data and generating DAO class files. I am excited to generate type safe queries and no HQL! In the package I provided please see the \*.sql files in the “sql” folder. These files create our data model.
2. Install docker if you don’t already have it installed. Note: If you are running on Apple Silicon you will have trouble running this container without Colima. Something additional I had to install on my end that allows X86\_64 containers to run on M1 ARM chips. On the command line I first launched colima with this command prior to executing docker…. colima start --arch x86\_64 --memory 4
3. Now run docker with the following command. This command will pull an Oracle XE 18.4 container from docker hub. It will initialize the database and create our NiQ Tables! You should substitute /Users/neil.jetty/Documents/niq/sql with the correct directory on your system where the “sql” folder resides... docker run --name NIQDB -d -p 1521:1521 -e ORACLE\_DATABASE=NIQDB -e ORACLE\_PASSWORD=temp1 -e APP\_USER=NIQ -e APP\_USER\_PASSWORD=temp1 -v /Users/neil.jetty/Documents/niq/sql:/container-entrypoint-initdb.d gvenzl/oracle-xe:18.4.0
4. You must wait for the database to fully initialize before continuing any further. Use command docker logs --follow <container ID> to follow the logs. Wait for “DATABASE IS READY TO USE” in the log outout.
5. The database is now running, with our schema created and waiting to service requests.

A screenshot of a computer

Description automatically generated

## Data Access Layer - jOOQ

I used jOOQ codegeneration tools to create Java classes for the tables in this database. Artifacts for this exercise are located in the “jooq\_codegen” folder. Java files were brought into the project manually. Interestingly jOOQ downloaded from Maven Central will not work with Oracle databases. I had to download a “trial version” of jOOQ for this to work and install it directly into my local maven cache.

## Spring Boot Web REST Service

The REST api developed here used Spring Boot. Main application code can be found in the PersonalDataApplication.java file. The solution was developed using Intellij IDEA. Do the following…

1. Launch the application
2. Jump to the command line and navigate to the folder “json”. We are now going to load all of the products into the database using the /addProducts endpoint designed to support the internal data team… curl -X POST -H "Content-Type: application/json" -d @./products.json http://localhost:8080/addProducts

A screenshot of a computer

Description automatically generated

1. Now let’s load the shopper shelves into the database using the other end point we created to support the internal data team (addShopperShelves)… curl -X POST -H "Content-Type: application/json" -d @./shelves.json http://localhost:8080/addShopperShelves
2. Next we will use the “getShoppers” endpoint designed for the external eCommerce server(s). In a browser execute this URL… http://localhost:8080/getShoppers?productId=BB-2144746857

A screenshot of a computer

Description automatically generated

1. Next we will use the “getProducts” endpoint designed for the external eCommerce server(s). In a browser you can execute the following URLs…

http://localhost:8080/getProducts?shopperId=S-1003

http://localhost:8080/getProducts?shopperId=S-1003&limit=2

http://localhost:8080/getProducts?shopperId=S-1003&category=Milk

http://localhost:8080/getProducts?shopperId=S-1003&category=Babies&brand=Childer

## Performance

To provide fast reads for the eCommerce operations, I added a number of indexes to the PRODUCT and PRODUCT\_SHELF tables. Each index was carefully considered in relation to the likely where clauses used by the two endpoints.

## Other things we could do to improve performance

1. If more performance was required (for example the number of records in the two tables were very high) once could consider partitioning the tables and/or indexes.
2. Expanding database memory so that more of the database is in memory rather than disk reads
3. Add read redundant replicas for the database layer with load balancing in front
4. Adding in memory cache (like redis) in front of the database layer
5. The application layer could be hosted in Kubernetes with HPA and VPA settings. This would also improve resiliency (high availability)
6. The application layer could be modified to use Spring Reactive Streams Framework with R2DBC drivers for the database. This would improve throughput for each web server node. The only drawback here is code readability and if you introduce a blocking call that is not using reactive streams, you could make performance worse.

## Productionizing these services

The code here is not production ready. There are a number of things needed here…

1. Database connection settings stored in an external key vault with scheduled password resets
2. The eCommerce end points should be brokered by an API gateway for consumer quota monitoring.
3. Authentication / Authorization solution in front of the eCommerce and internal APIs.
4. A monitoring solution for real-time alerts on performance and/or service degradation
5. Add more logging and logging levels inside the application layer with log consolidation in the backend with efficient search capabilities.