DBD Exam - Bilbakken

Assignment Link

Github link:
https://github.com/SOFT2021-Databases-For-Developers/Exam-Project

Enterprise:
Car Marketplace

Author Details:
Jonatan Magnus Klarskov Bakke (cph-jb281@cphbusiness.dk)
Jonas Valentin Björk Hein (cph-jh365@cphbusiness.dk)
Thomas Stevns Nielsen Ebsen (cph-te52@cphbusiness.dk)

Contents

Introduction

For our Database exam we have chosen to make an online car dealership called Bilbakken in the style of bilbasen.dk.

Our car dealership consists of four different types of databases: Neo4j, Redis, PostgreSQL and MongoDB.

MongoDB is a NoSQL document based database, which we use for storing our users and their orders.

PostgreSQL is a relational database, which we use for storing the makes, models, cars (composed of make and model) and Listings.

Neo4j is a graph database, which we use for storing recommendations for the users based on the listings the user has seen.

Redis is a NoSQL key-value store, which we use for storing the shopping cart of a user. The last type of database we have is the key–value NoSQL database.

User/Order database	MongoDB
Product/Listing database	PostgreSQL
Recommendation database	Neo4j
Session/Shopping cart	Redis

The way the site is meant to work, is that a user can log in on the site, from that point on the user can choose to make a listing of a car they want to sell, see other users listings, see recommendations based of what types of listings the user have clicked previously, or add a listing to the shopping cart. When the user has a shopping cart they then can choose to checkout and “buy” the listing.

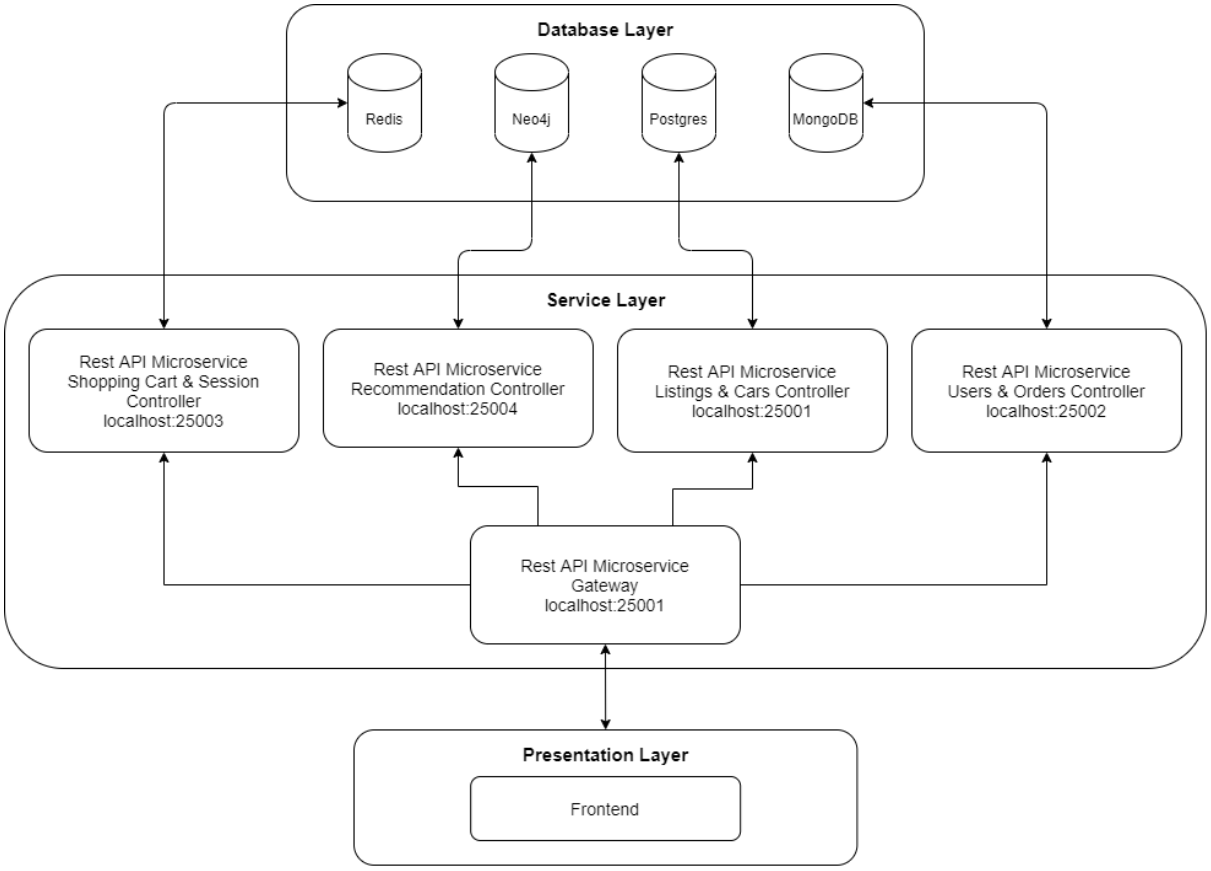
Functionality
A user should be able to register and login
A user should be able to retrieve a list of cars that are being sold
A user should be able to see a specific listing
A user should be able to add, remove or update their car listings
A user should be able to add listing to their shopping cart
A user should be able to purchase a list product
A user should be able to see recommendations

Setup

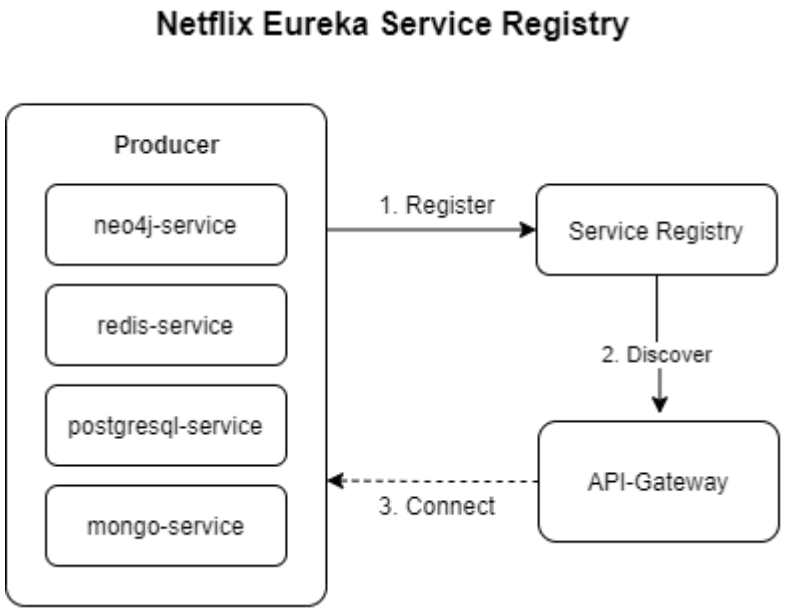
Project diagram

We have chosen to use the microservice architecture to streamline and organize the calls to our different types of databases. To achieve the microservice architecture, we split all of the different rest API into their own specific microservice, whose only job is to communicate to one specific database.

The microservices are then all connected to the rest api gateway through eurekas discovery server, so that they can be queried through the api-gateway.



The diagram shows how we utilize Netflix Eureka to register our four rest api's into our Service Registry, so that the api-gateway can discover them



Mongodb

Users and orders

This application uses a MongoDB database to store the user of the application and the orders that a user makes. The reason we have chosen to use MongoDB to store our users and orders is because of the horizontal scaling options that MongoDB have. The scaling options will give great benefits, with an ever growing backlog of order to store. We have also chosen MongoDB because of its document structure which provides easy storing of data while also providing solid querying options.

Data

The data that composes a user in our application is as follows:
String id, String firstName, String lastName, String email, String password.

Orders are composed of:
String id, Date createdAt, UserDTO user, List<Integer> listingIds.

Api

By using the Gateway you can interact with the Rest Api containing the MongoDB part by calling:

HTTP Method	URL	Body	Response
GET	localhost:25000/orders		ResponseEntity<Collection<OrderDTO>>
GET	localhost:25000/orders/{id}		ResponseEntity<OrderDTO>
GET	localhost:25000/orders/user/{id}		ResponseEntity<Collection<OrderDTO>>
POST	localhost:25000/orders	User	
DELETE	localhost:25000/orders/{id}		

HTTP Method	URL	Body	Response
GET	localhost:25000/users		ResponseEntity<Collection<User>>
GET	localhost:25000/users/{id}		ResponseEntity<User>
GET	localhost:25000/users/{id}/listings		ResponseEntity<Collection<Listing>>
GET	localhost:25000/users/{id}/orders		ResponseEntity<Collection<OrderDTO>>
POST	localhost:25000/users/login	User	
POST	localhost:25000/users	User	

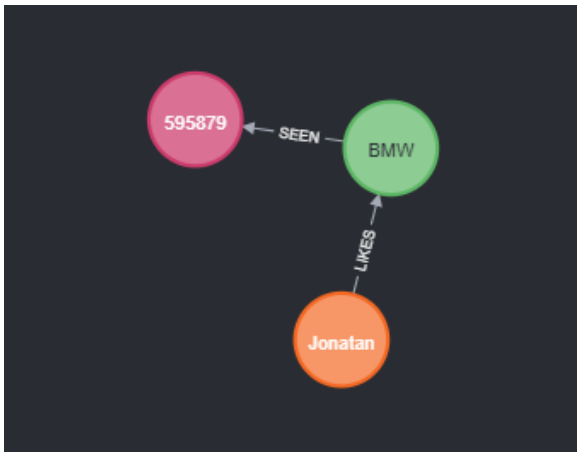
Recommendations/Neo4j

Recommendations for the users

To present our users with valuable information we created a recommendation engine to help the user find cars that are similar to other cars the user has been looking at. This functionality is very common among online webshops and we wanted to make it as easy for the user as possible to quickly find cars that the user would prefer over other cars. To do this we needed to store information about the user as in what listings they have been looking at and present this data to the user in a useful way. There are many ways to do this and we landed on saving information about what brand of cars the user liked looking at. So if a user looks at five BMW's and two FORD's they will be recommended BMW then after FORD, since they have looked at more cars from BMW than FORD.

Data

In our neo4j database we have created three node types to store information, we have Person, Make and Listing. From these three nodes it is possible to give the users their recommendations.



The Person node holds unique information about the user to be able to identify them from the rest. This could be the user email or an id of some sort. The Make node holds information about the car's brand. The Listing node holds information about the Listing, the data saved for a Listing is the listingId. The Person node has a unidirectional relationship called LIKES connecting to a Make node. The LIKES relationship is used to move from Person node to Make node. The Make node has a unidirectional relationship called SEEN connecting to a Listing node. The SEEN relationship is used to move from Make node to Listing node. The total path from a Person node to a Listing node in cypher looks like:.

```
(p:Person)-[:LIKES]->(m:Make)-[:SEEN]->(l:Listing)
```

When generating recommendations for the user we read through the data model and create a Recommendation. A Recommendation is a POJO that contains the make and a count. The make is the brand of the car and count is the number of times the user has clicked a listing for the specific make. To get the data necessary for a Recommendation we created this cypher quarry:

```
MATCH(n:Person{name:$name})-[]->(m:Make)-[]->(l:Listing) RETURN {make: m.make, count: count(l)};
```

This quarry uses the username to find the Person node in question and via the relationships LIKES and SEEN finds the Make and Listing nodes. When we have this information we create a result object containing the make name and the number of Listings attached to the Make node. If there are more than one Make node for a Person then it will create a list of the result object. This makes it very easy to extract the information from this quarry and create our Recommendation list. The list works in the way that the object in the list with the highest count is the car brand we recommend the most to the user.

Api

By using the Gateway you can interact with the Rest Api containing the neo4j part by calling:

HTTP Method	URL	Body	Response
POST	localhost:25000/recommendation	RecommendationPost	
GET	localhost:25000/recommendation/{username}		List<Recommendation>

Session/Redis

Session and Shopping cart

When a user logins to our application we create a session with information about the user. The session holds the unique identifier provided by the user and saves it in our session alongside adding a shopping cart. To give the user the ability to look at more than one listing at a time we decided to add a shopping cart to the session. So if a user closes the application the session is saved, so next time they use the application the session remembers the user's shopping cart information.

Data

In our redis database we have a very simple way of saving the session as a SessionObject. Our SessionObject holds the unique identifier based on the user and the shopping cart as a List containing the Listing Id's. When we persist the SessionObject in the redis database we use the unique identifier as the key and the SessionObject as the value.

KEY	VALUE
"Jonatan"	"{username: 'Jonatan', shoppingCart: [{ '32324' }, { '55445' }] } "

Api

By using the Gateway you can interact with the Rest Api containing the redis part by calling:

HTTP Method	URL	Body	Response
POST	localhost:25000/session	SessionObject	
GET	localhost:25000/session/{username}		SessionObject
DELETE	localhost:25000/session/{username}		

PostgreSQL

We have chosen to use PostgreSQL for storing the components which the application uses to comprise a car listing. We have also chosen to store in PostgreSQL our Car element which is composed of Make and Model Class and an id. Make contains the car brand e.g BMW which is used in Model to acquire the right models from the chosen Make brand which contains the name of a model and the year. What a car listing is composed of is, the aforementioned Car, and the remaining data in the listing is seller of the listing, title, description, price, km, status and the created_on date.

The reason we choose to use a PostgreSQL database to store the car and listing related data is because of the benefits we gain from having well defined relations.

Those relationships make it easier when retrieving the right car models from Make, which would have been a more difficult process without the relations we gain from PostgreSQL.

Api

By using the Gateway you can interact with the Rest Api containing the PostgreSQL part by calling:

HTTP Method	URL	Body	Response
GET	localhost:25000/cars		ResponseEntity<Page<Car>>
GET	localhost:25000/cars/all		ResponseEntity<Collection<Car>>

GET	localhost:25000/cars/{id}		ResponseEntity<Car>
POST	localhost:25000/cars	Car	
PUT	localhost:25000/cars/{id}	Car	
DELETE	localhost:25000/cars/{id}	PathVariable("id")	

HTTP Method	URL	Body	Response
GET	localhost:25000/makes		ResponseEntity<Page<Make>>
GET	localhost:25000/makes/all		ResponseEntity<Collection<Car>>
GET	localhost:25000/makes/{id}		ResponseEntity<Make>
POST	localhost:25000/makes	Make	
PUT	localhost:25000/makes/{id}	Make	
DELETE	localhost:25000/makes/{id}	PathVariable("id")	

HTTP Method	URL	Body	Response
GET	localhost:25000/models		ResponseEntity<Page<Make>>
GET	localhost:25000/models/all		ResponseEntity<Collection<Model>>
GET	localhost:25000/models/{id}		ResponseEntity<Model>
POST	localhost:25000/models	Model	
PUT	localhost:25000/models/{id}	Model	
DELETE	localhost:25000/models/{id}	PathVariable("id")	

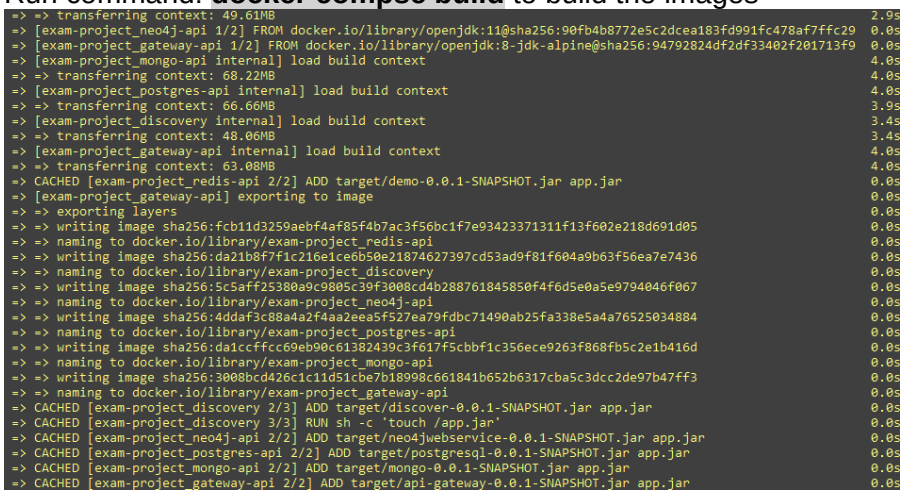
HTTP Method	URL	Body	Response
GET	localhost:25000/listings		ResponseEntity<Page<Listing>>
GET	localhost:25000/listings/make/{name}		ResponseEntity<Collection<Listing>>
GET	localhost:25000/listings/{id}		ResponseEntity<Listing>
POST	localhost:25000/listings	Listing	
PUT	localhost:25000/listings/{id}	Listing	
PUT	localhost:25000/listings/{id}/status/set/{status}	PathVariable int id, PathVariable Status	

How to run the project

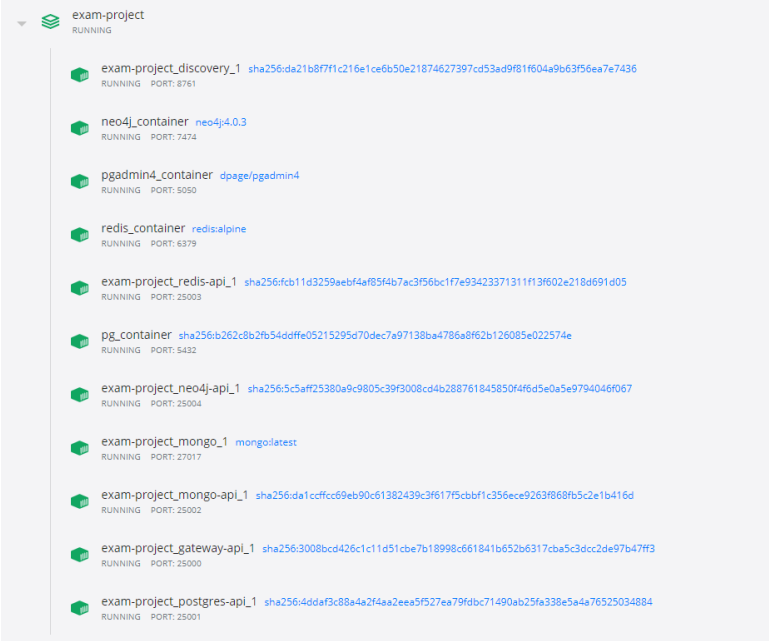
Requirements: Docker and working linux kernel

To run the whole application, open a CMD inside of the root of the project.

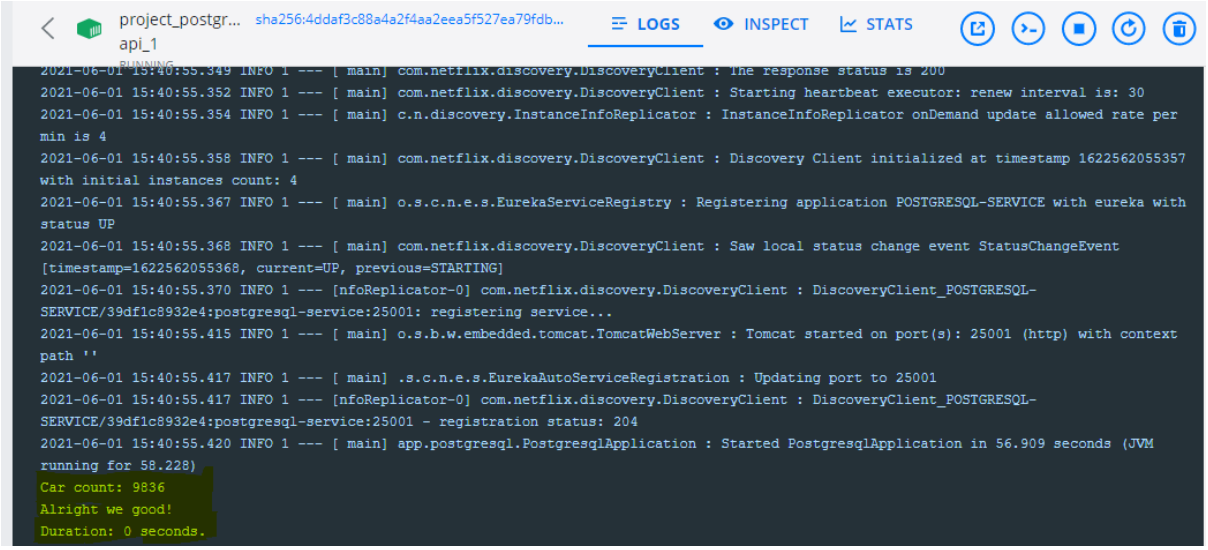
1. Run command: **docker compse build** to build the images



2. Run command: **docker compose up** to setup and start the project



3. The setup may take some time to complete due to the large amount of cars we insert into postgresql, so make sure it has finished loading before querying. A sure way to know that has finished setting up (assuming docker didn't crash) is to click on **exam-project-postgres-api_1** to see the console.



It's finished setting up once you see the **"Alright we good"** message in the console as pictured above.