DB Exam Project 2020

Group: #TypeScriptCarry

- Mohammad Hariri
- Rasmus Jarnborg Friis
- Andreas Guldborg Heick

Exam Description

In the modern IT world, many businesses provide their products or services on-line, using Internet as a channel of communication with their customers. During economically hard times as Corona isolation for many it remains the only platform for marketing and retailing.

To meet the requirements of the on-line digital transition, modern applications work with purpose-built and decoupled data stores, where different application components can be mapped to different data sources, rather than to a single database. This approach is implemented in microservices architecture, as well as in polyglot persistence applications.

Project Objective

Your objective is to provide, as a proof of concept, database support to an application or related

applications, where multiple diverse database models are built and implemented for different purposes.

Table of contents

Installation Instructions

Requirements:

- Install all required databases (Neo4j, MongoDB, PostgreSQL)
- Node

1. Open and start databases - make sure constants.ts in the root folder matches your login credentials for all databases
    a. PostgresSQL: CREATE DATABASE db_exam;
2. Cd into the project root
3. npm install
4. npm start

If you get errors or don't get "[Database name] is connected" message in your console for each database, then you probably didn't start the database services correctly.

Requirements

Possible use case:

For a blog or social media, where data is the main business model.
With our application a media could analyse relations between users and use this to improve the platform. From these relations we would know who follows who, and then we can look at posts and show the posts of all followed.

Logs would provide us with data about usage so we can improve, and it would help with security. With sessions and authentication a media could make sure who can use and see what.

Functional requirements:

1. Authentication - Users have to be authenticated to use the application

2. Database Security - only people with authenticated direct access or access through the API should have access to the databases.

3. User Sessions - Users sessions needs to be stored so we at all times know who is logged in and if they are authenticated to use the application.

4. Data Analysis - We need a way of analyzing users and their relations to other users.

5. Storage of data - We need a place to store Users and Posts.

6. Logging - We need a way of logging interactions and use of the application, and a log storage so we can go back and view the usage. We also need logging for security purposes, to be able to catch unauthorized usage.

Non-functional requirements:

1. Has to work in Postman

2. Has to work on Windows 10 & Mac

Plan

We have chosen to build a polyglot persistence application that implements and uses four different databases. We have chosen Neo4j, MongoDB, Redis and PostgreSQL to build our application.

The idea behind the application is to make a social media using four different kind of database each to their advantages.

To store user data we want to use Neo4j to store the users, and the users relation with each other. A relation in our application is going to be following one or multiple users. We want to use Neo4j because it’s super easy to run algorithms on data, and easy to query and find.

For posts we are going to use MongoDB to store posts that the users make, the idea here is that a user will only be able to see posts, from people they follow.

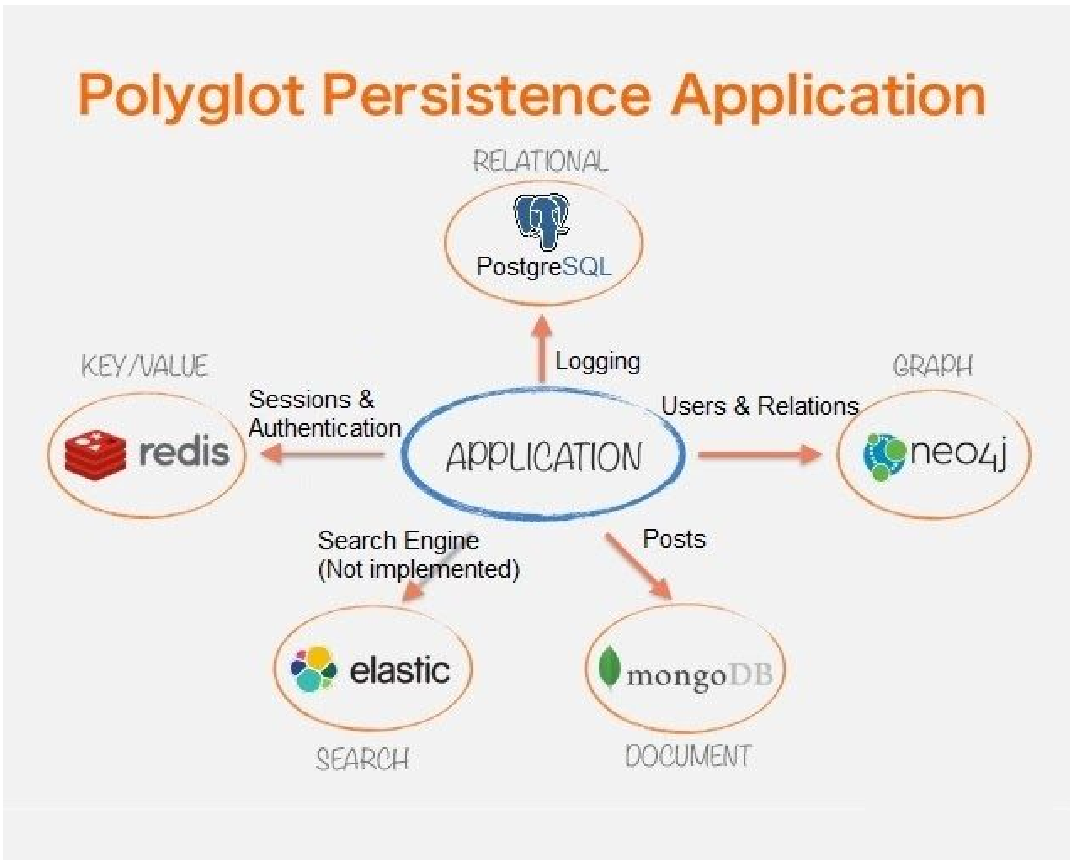
We also want people to login, so we can control access and hide endpoints behind a secure wall. For this purpose we will use Redis. Redis will control the users sessions and general authentication and we plan to make it so everything needs to be authenticated before we allow interactions with any of the databases.

Our plan for database security is to protect the databases from direct access attacks, meaning that to access our databases directly a person would need the correct login information. The only other way of talking to the database will be by going through the API, and all critical endpoints here should therefore be protected. So for API access a normal user would have to log in.

Lastly we are thinking about using PostgreSQL to control the logging of the system, because it is an important security aspect, and we can use it to see usage and maybe improve the platform. Logging can also be an essential part of security, used to spot issues and solve them.

Finally if the goal was to create our application as a production full polyglot persistence application, we found that for searching through large amounts of data, an application like ours could benefit from a search engine like Elastic Search. Elastic Search is an open-source search engine that can store documents, or in our case users or posts. Elastic Search is great for databases with huge amounts of data, where fast and good searching on the platform is required. We won't be using Elastic Search but it could be fun to try and incorporate.

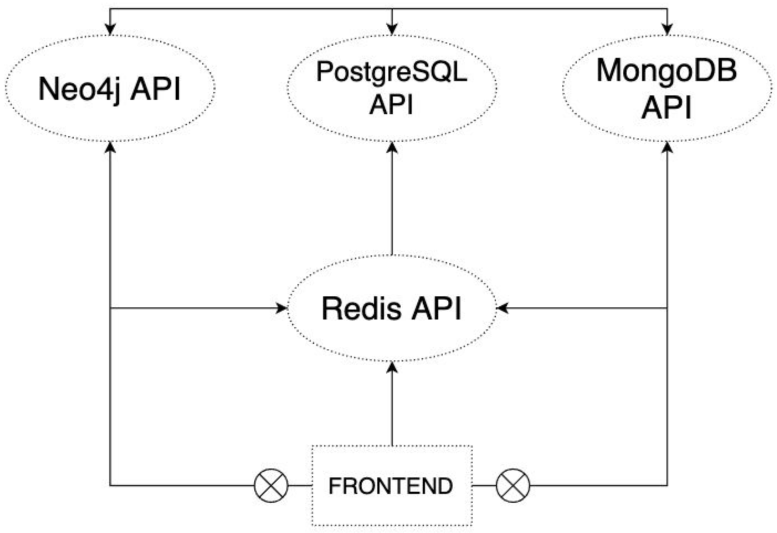
Overview of our application plan



Final Product

Authentication & User Sessions

To authenticate and manage user sessions while still implement microservice architecture, we used Redis. Everytime someone sends a request to our API we make sure it pass through the Redis API in the beginning, this is done by making sure every request has its origin from the Redis API. This is shown in the diagram below.



The Redis API does all the authentication, by checking the request for a valid token, then verifying its structure and checking it up against the Redis database. If the request token matches the Redis database token, then the user is authenticated. The sessions stored in the Redis database, contain a username (key) and a token (value).
All our key-value pairs in Redis have a TTL of 12 hours, after which it expires and is deleted from the memory. An example of using Redis to store tokens can be seen in Example 2 - Register Method in the appendix

Database Security

All our databases have been setup with login required to access the databases. Neo4j, MongoDB and PostgreSQL are all local databases, so we used basic unsecure login credentials so all of us could easily access them. In reality we would have used real users and passwords, that couldn't be guessed as easily.
Redis is the only database that is hosted in the cloud, so we connect to it using a connection string. This isn't the most secure solution but it still requires the unique connection string.

All of this means that our databases are secured and require login for direct access or access through the use of the API.

Users & Followers

We used Neo4j to store our Users and the relationship Follows, so we can keep track of which users that follow which, and so we can store Users. To interact with our User and Follow data we made several endpoints so we can create Users, find them or follow other users. You can see how the create works by looking at Diagram 1 - Register a user and Example 2 - Register Method in the appendix.

We also used Neo4j to analyze our data and automate this process by creating REST endpoints that return the results of our algorithms and analyzation queries. We created two endpoints for this purpose, one uses ArticleRank on our data and returns the top 10 most important users, the other uses MATCH to return top 10 most followed users.

Below is the Article Rank query. ArticleRank is a variant of the Page Rank algorithm. ArticleRank differs from Page Rank, by lowering the weight of low out-degree. This means that in Page Rank if a user follows no one and is followed by many, then the user scores high. But if the user follows himself then he also gets a boost, because he is only following a very followed person who doesn't follow others.
ArticleRank fixes this issue by lowering the score boosts, and this gives us the best result by not giving the first place to someone only because the user followed themself. That is why we chose to use ArticleRank.

```
CALL gds.alpha.articleRank.stream({
  nodeProjection: 'User',
  relationshipProjection: 'FOLLOW',
  maxIterations: 20, dampingFactor: 0.85 })
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).username AS name, score
ORDER BY score DESC, name ASC LIMIT 10
```

The query below uses MATCH to find the top 10 most followed users. It finds all Users and counts all relations that points to the User.

```
MATCH (n:User)<-[r]-(x)
RETURN n, COUNT(r)
ORDER BY COUNT(r) DESC
LIMIT 10
```

Posts

We are using MongoDB to store all the posts that a user makes. The reason we are using MongoDB for this action is because the posts do not need a relation, the only thing we need to store about a post is the data for the post content itself, and who made it.

The REST endpoints we made for this is to create a post, find a post by either it's ID or the owner's username or just finding all posts.

Logging

We are using PostgreSql to save logs about which HTTP method is used when a user sends a request to a REST endpoint, and what they are sending as body. The reason we store these logs, is because it can help the owner of the system to see activities of its users, and discover potential security issues or stop attacks. A log is created each time one of the Rest endpoints of the system is getting called, and saved in our PostgreSQL database.

To access the logs we made a route to get all the logs, and a route to get all logs from the last twenty four hours ago in the API. We made the last route because getting all the logs ever created can get overwhelming really fast, and mostly an admin would only need logs of the last 24 hours.

You can see how logging works by looking at Diagram 1 - Register a user and Example 1 - Register a user in the appendix.

Technologies

Databases:

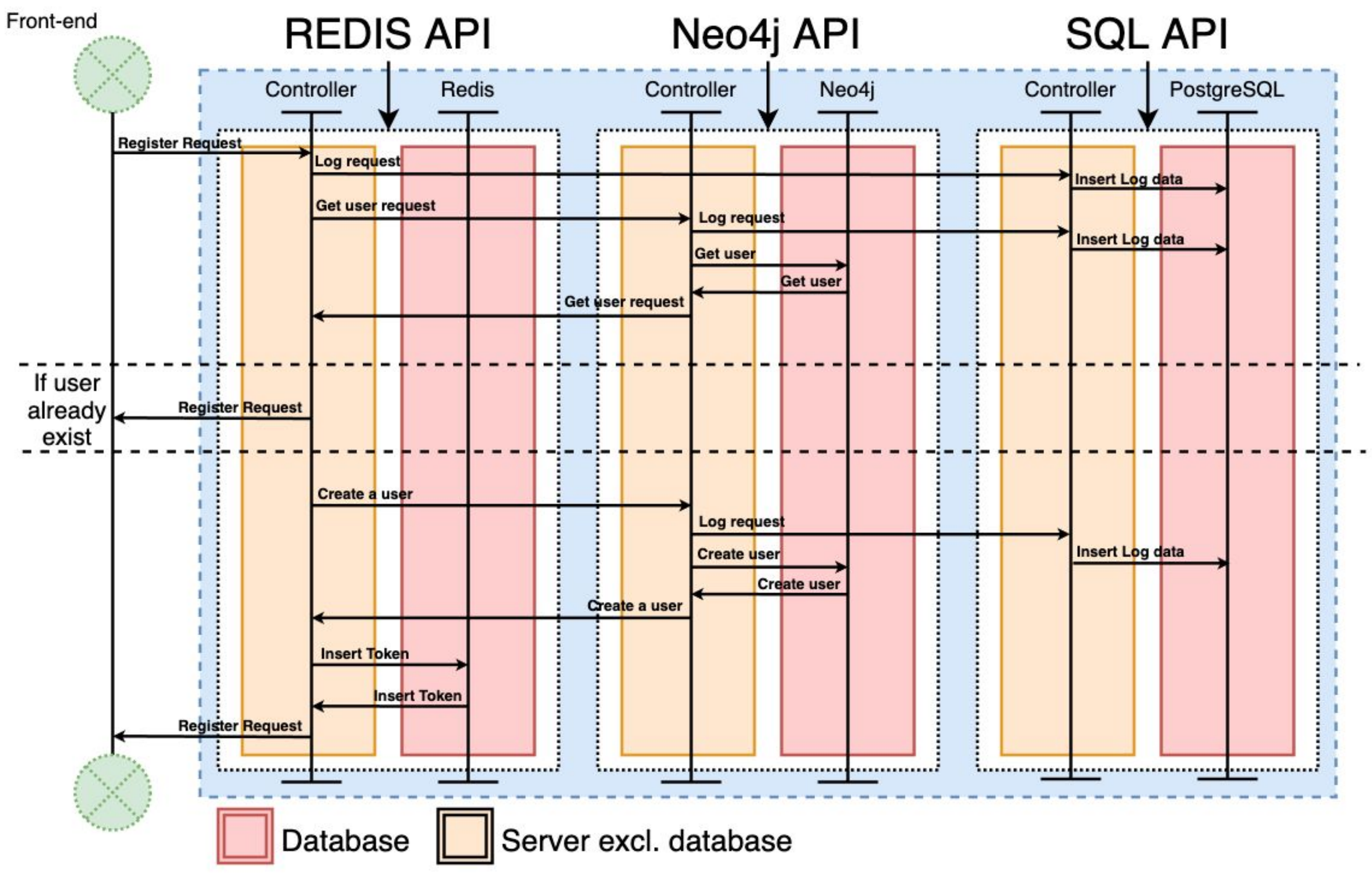
- Neo4j - NoSQL graph database
- MongoDB - NoSQL document-oriented database
- Redis - key/value NoSQL database
- PostgreSQL - relational SQL database

Tools & languages:

- Node - JavaScript runtime for server side scripting
- ExpressJS - Node web application framework for building web applications and API's
- TypeScript - A strict syntactical superset of JavaScript that adds optional static typing. Usually used for larger applications.
- Postman - A tool to test and develop API web requests

Appendix

Diagram 1 - Register a user



Example 1 - Register a user

This route is to register a user. This is an example where we user 3 databases on a single POST request.

```
route.post('/register', async (req, res, next) => {
  const { username, password } = req.body;
  log(req.method, req.body);
  try {
    const token = await authenticationLogic.register(username, password);
    if (!token) return res.sendStatus(400);
    return res.json(token);
  } catch (ex) {
    console.log(ex);
    return res.sendStatus(500);
  }
});
```

Logging with PostgreSQL

What happens here is the log method is calling a function that logs what happens and saves it in our PostgreSQL database. Below is the method register in detail.

Example 2 - Register Method

```
register = async (username, password): Promise<string> => {
      const user = await
get(`${NEO4J_API}/user/find/username/${username}`).then(response=> {
      return response.json();
      });
      if (Object.keys(user).length > 0) return undefined;
      const isCreated = await post(`${NEO4J_API}/user/create`, { username,
password }).then(response => {
      return response.status === 200;
      });
      if (!isCreated) return undefined;
      const token = tokenService.create(username);
      sessionMapper.set(`name:${username}`, token);
      return token;
  };
}
```

What happens here in short is that the function register get called from the Redis API. The function then checks in the Neo4j database if the user already exists. If the user exists then the function just returns, however if the user does not exist then the /user/create endpoint is called in the Neo4j API, which is where a user is created.

When a new user is created, then user is also added in a Redis database, with the user's username, and a token that granted access to the system as a key-value pair. The pair is delete after 12 hours in Redis, meaning after that, the user has to login again.