WORKPLACE ORGANIZATION APPLICATION

Technical Design Document

Fontys ICT
Tsanko Nedelchev

Contents

INTRODUCTION 2

ENTERPRISE SOFTWARE PLATFORM 3

    Range of applicability 3

    Development Speed 3

    Manageability and Flexibility 3

ARCHITECTURE OVERVIEW 4

MICROSERVICE ARCHITECTURE 6

MICROSERVICES USER REGISTRATION COMMUNICATION VIA RABBITMQ 7

MICROSERVICES AUTHENTICATION AND AUTHORIZATION COMMUNICATION 8

RABBIT MQ 9

MONGODB DATABASE 10

CONCLUSION 11

INTRODUCTION

This technical design document provides an overview of the system architecture, design, and implementation details for the "Workplace Organizer Application". The application consists of multiple microservices, including a user authentication and authorization service, a meeting scheduling service, a notifications service, and a meeting notes service. Each microservice is designed to handle a specific aspect of the application, with communication between the microservices being managed through RabbitMQ messaging. The application will be developed using TypeScript, Node.js, Express, MongoDB, and Mongoose. The aim of this document is to provide a high-level understanding of the architecture and design of the application, as well as to outline the key features of each microservice.

Furthermore, an overview for each technology in the tech stack will be available in order to justify why it was chosen for this project and what it accomplishes.

ENTERPRISE SOFTWARE PLATFORM

The enterprise software platform is one of the most important choices to make when creating an enterprise software application. The criteria that will be looked at in this section include – Range of applicability, Development speed, and manageability and flexibility.

Range of applicability

The range of applicability of a development platform depends on the capabilities of said platform. Typescript with NodeJS and Express is a very universal tech stack that is often used together to create microservice applications. The platform-agnostic nature of TypeScript with Node.js and Express enables deployment on diverse operating systems, such as Windows, Linux, and macOS. Consequently, the individual API microservices can be conveniently deployed on different platforms and operating systems, ensuring flexibility and compatibility.

In terms of scalability the NodeJS, Typescript, Express tech stack performs very well thanks to NodeJS's support of async development and parallel communication between services. Typescript enhances code reuse through its templating features, although it demands substantial initial setup time. Node.js relies on the V8 JavaScript engine, recognized for its efficiency and performance. With its modest memory requirements and minimal overhead, Node.js adeptly manages concurrent connections and scales horizontally by deploying numerous microservice instances across multiple servers.

Development Speed

TypeScript provides a strong type system, which helps catch errors during development and improves code quality. It offers great tools support with integrated development environments such as Visual Studio Code, enabling efficient code editing, debugging, and testing. It's static typing and autocompletion capabilities with the help of some third party software and IDE plugins speed up the development process by a lot once everything is setup. Furthermore, TypeScript with Node.js and Express benefits from a huge array of third-party libraries and components, which accelerates development by providing ready-to-use solutions for the most common functionalities.

Manageability and Flexibility

Typescript with NodeJS and Express introduces some overhead at the start as it requires some setup, however once the project structure and workflow is setup things start to puck up quickly. TypeScript supports modularity and follows the CommonJS module system, allowing for the organization of code into reusable and maintainable modules. Express provides middleware-based architecture, which enables easy modification of the application's behavior.

ARCHITECTURE OVERVIEW

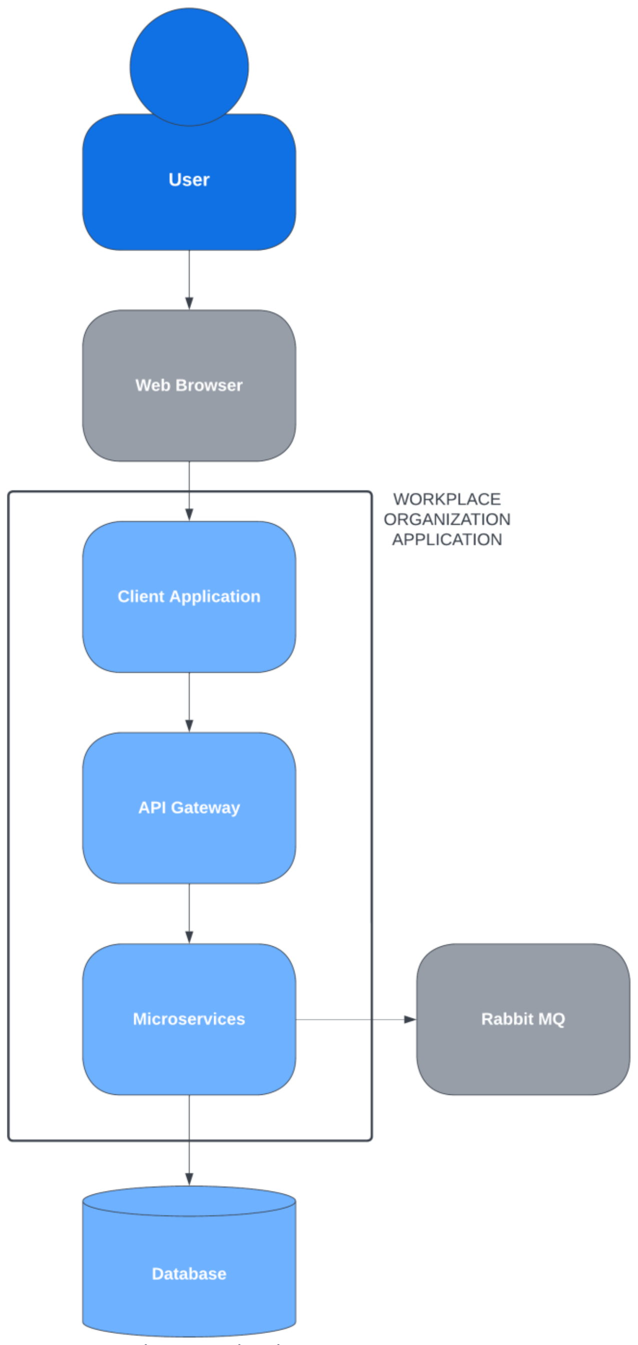


Figure 1 - C4 diagram - level 2

The application's architecture is designed to be a microservices-based architecture, consisting of several microservices that interact with each other via Rabbit MQ messaging. A gateway is used to manage the external API interface and to route requests to the appropriate microservices.

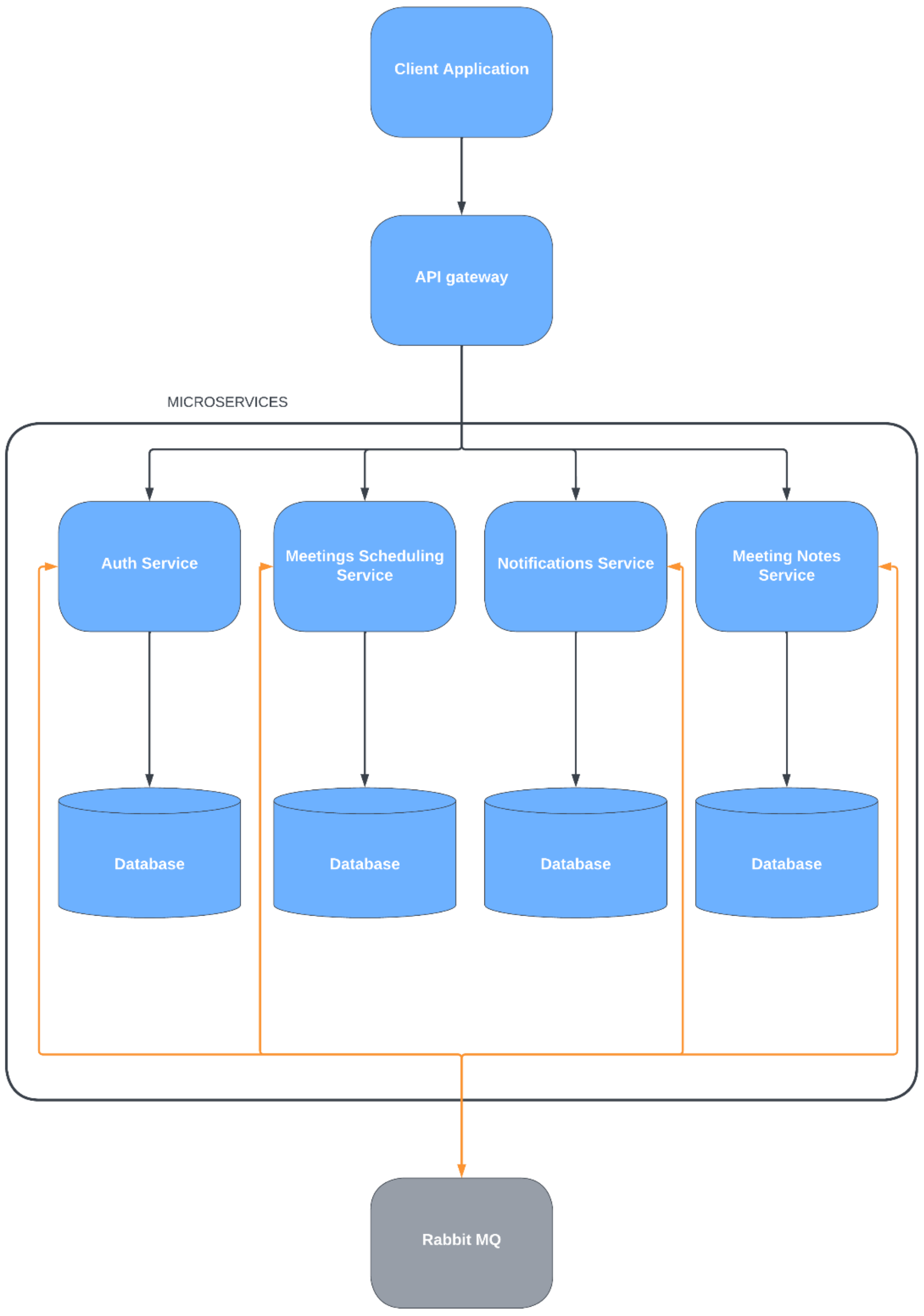


Figure 2 - C4 diagram - level 3

The back end system consists of the following microservices:

Authentication and authorization microservice - provides user authentication and authorization functionality, including login, logout, and access control.

The Meetings Scheduling microservice - provides functionality for scheduling meetings, including creating, updating, and deleting meetings.

The Notifications microservice - responsible for sending notifications to users when they have upcoming meetings, as well as sending notifications for other system events, such as new team invitations.

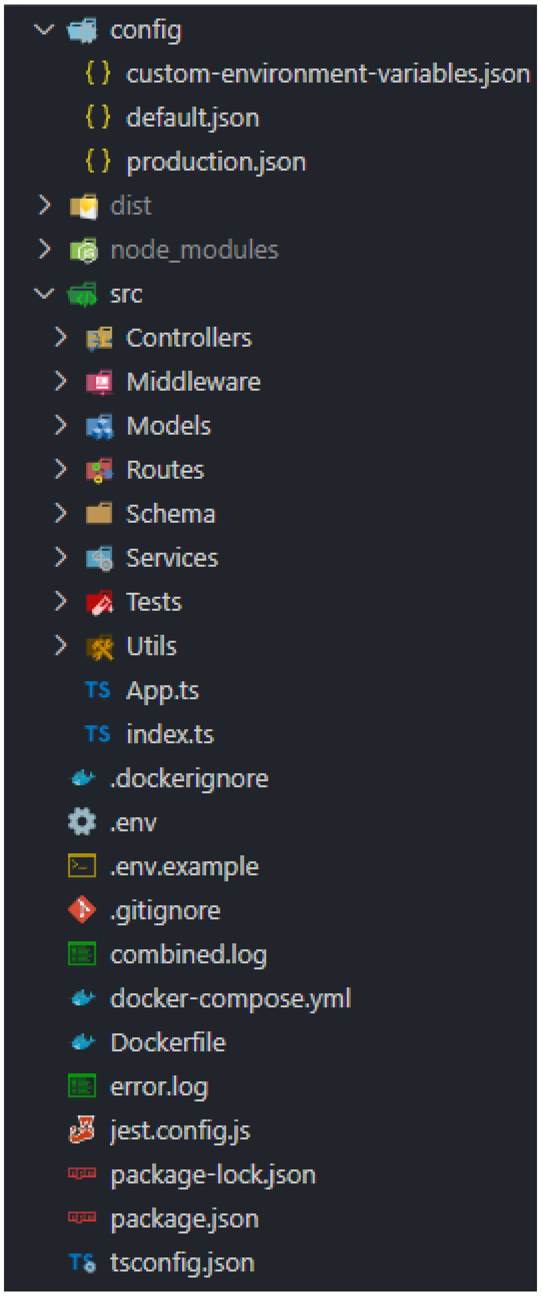
The Meeting Notes microservice - provides functionality for creating and editing meeting notes, as well as storing and retrieving them from the database.

MICROSERVICE ARCHITECTURE

Each microservice is designed to be loosely coupled and independently deployable which allows easy scaling and maintenance. The microservices interact with each other via Rabbit MQ – a message broker, with each microservice consuming messages from another to perform tasks.

The microservices are implemented using NodeJS and TypeScript with the Express framework.

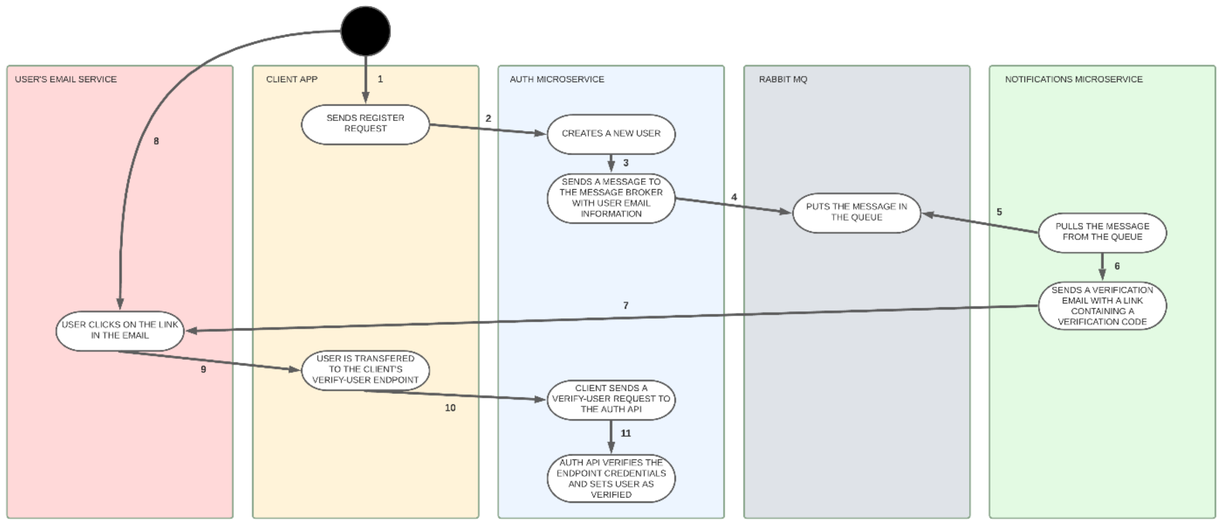
Folder structure and microservices setup



```
config
    custom-environment-variables.json
    default.json
    production.json
dist
node_modules
src
    Controllers
    Middleware
    Models
    Routes
    Schema
    Services
    Tests
    Utils
    App.ts
    index.ts
.dockerignore
.env
.env.example
.gitignore
combined.log
docker-compose.yml
Dockerfile
error.log
jest.config.js
package-lock.json
package.json
tsconfig.json
```

As an example this is the auth microservice. It is comprised of the base directory where the **package.json** file is located along with other config files such as the **tsconfig.json** file which handles the typescript rules for the project, the **jest.config.js** file which handles the path to the test folder and specifies the way tests are to be performed, the Dockerfile which specifies the way the Docker image of the microservice is to be built and the environment variable files – the .env which holds the environmental variables for the local development process and the example .env file which serves a purpose inside the git repository by describing the types of variables that should be present however it doesn't contain any values for them.
The config folder contains several config files one for local development and one for production. The "**custom-environment-variables.json**" is a setup file that takes the env variables from .env and shares them with the "**default.json"** and the "**production.json"** .
And finally the src folder where the main part of the code resides. The index.ts is the starting point of the project. It imports an instance of App.ts which contains a class of the Express server.
The Schema, Models, Services, Controllers, Middleware contain the schema, model, service ,controller and middleware levels code respectively.
The Routes folder defines the express routes of the API. The Tests folder contains the Jest unit tests of the service layer functions only and the Utils folder contains different utility functions like JWT generation and verification, RabbitMQ connection functionality, MongoDB connection functionality, etc.

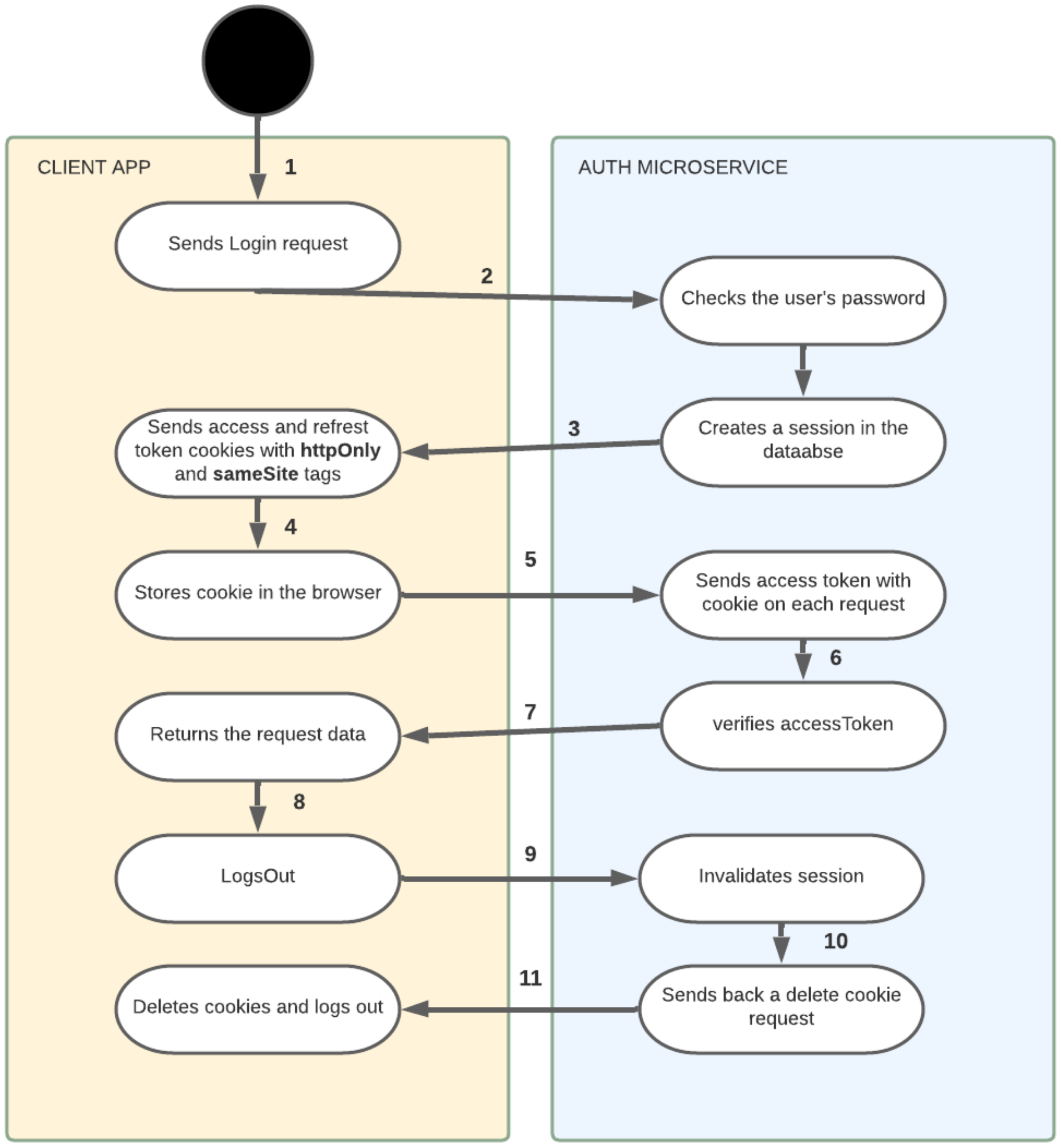
MICROSERVICES USER REGISTRATION COMMUNICATION VIA RABBITMQ


USER'S EMAIL SERVICE
CLIENT APP
AUTH MICROSERVICE
RABBIT MQ
NOTIFICATIONS MICROSERVICE
1
SENDS REGISTER REQUEST
2
CREATES A NEW USER
3
SENDS A MESSAGE TO THE MESSAGE BROKER WITH USER EMAIL INFORMATION
4
PUTS THE MESSAGE IN THE QUEUE
5
PULLS THE MESSAGE FROM THE QUEUE
6
SENDS A VERIFICATION EMAIL WITH A LINK CONTAINING A VERIFICATION CODE
7
USER CLICKS ON THE LINK IN THE EMAIL
8
9
USER IS TRANSFERED TO THE CLIENT'S VERIFY-USER ENDPOINT
10
CLIENT SENDS A VERIFY-USER REQUEST TO THE AUTH API
11
AUTH API VERIFIES THE ENDPOINT CREDENTIALS AND SETS USER AS VERIFIED

The user tries to register a new account the request is sent to the auth microservice and it creates an account int eh database. Then, through Rabbit MQ it sends the event to the queue. From there a notifications-microservice is listening for any events from RabbitMQ. The moment a new even arrives the notifications microservice pulls it from RabbitMQ and sends an email verification containing the message which is a user verification URL with credentials params to the email of the user. The user can now check their email and find the verify user link in their inbox. When they click the link they are redirected to the client application which automatically sends a request to the auth-microservice to verify the user email address. The email address is verified and now the user can log into their account.

If the user registers using the SSO with their Google Account then the verify email address functionality and responsibility is transferred to Google.

MICROSERVICES AUTHENTICATION AND AUTHORIZATION COMMUNICATION


CLIENT APP
AUTH MICROSERVICE
1
Sends Login request
2
Checks the user's password
3
Creates a session in the dataabse
Sends access and refrest token cookies with **httpOnly** and **sameSite** tags
4
Stores cookie in the browser
5
Sends access token with cookie on each request
6
verifies accessToken
7
Returns the request data
8
LogsOut
9
Invalidates session
10
Sends back a delete cookie request
11
Deletes cookies and logs out

Use case chart of a user logging in and logging out of the WOA application. The httpOnly tag on the cookie disables the cookie from being manipulated from the client side and instead it can only be altered by the server. The sameSite tag means that the client and the server should be on the same domain otherwise the cookie doesn't work. After the client receives the cookie it can send it to the postfeed-microservice (notes microservice) as well and it will automatically check if the accessToken is valid and if it is the user is authorized to use the postfeed-microservice (notes microservice) too. If the user chooses to use SSO with their google account they are logged in automatically.

RABBIT MQ

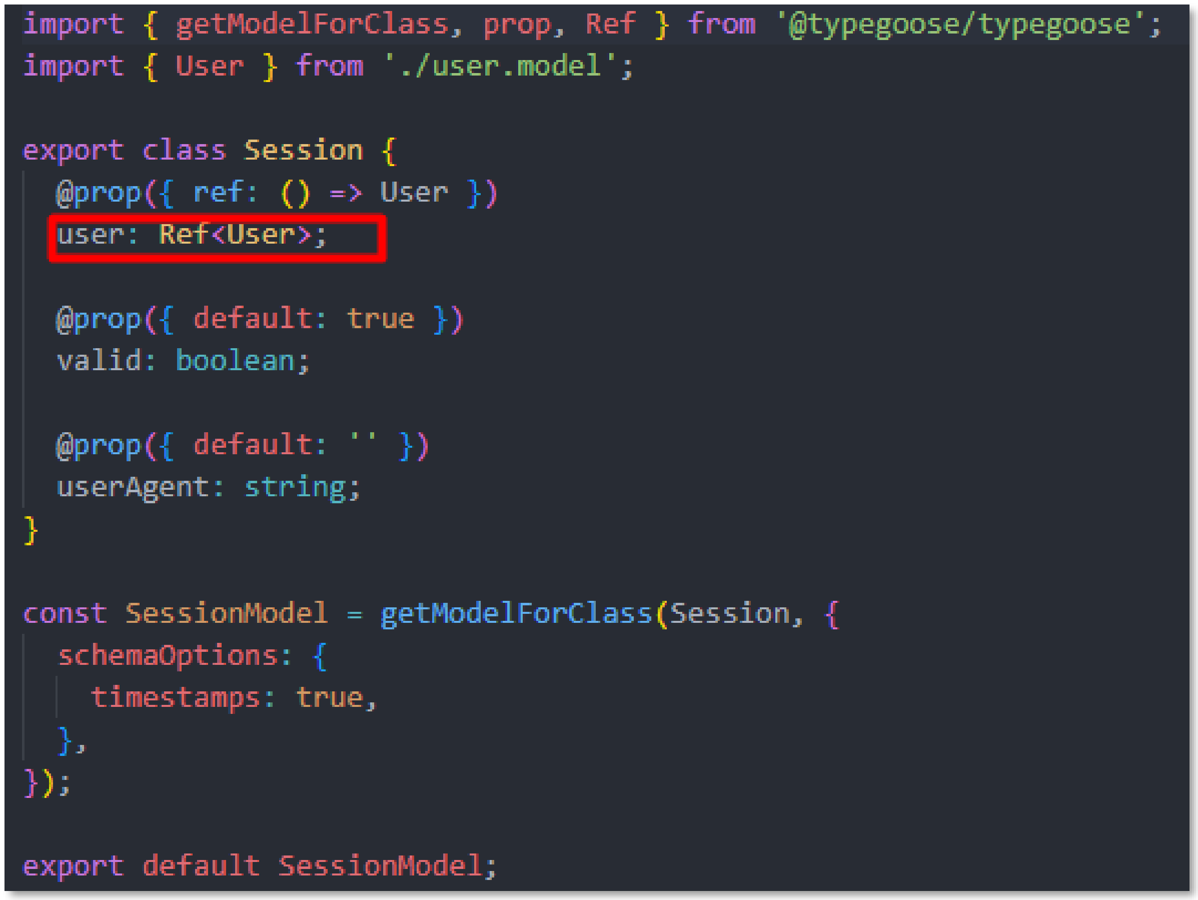
RabbitMQ is a popular open-source message broker that provides a flexible and reliable messaging system for applications. It provides a lot of different functionalities that have to do with messaging however for this application use case RabbitMQ acts like a message queue between the authentication microservice and the notifications microservice. The auth-microservice instance is the publisher that publishes events to the queue with the registered users email addresses and verification credentials and the notifications-microservice consumes the messages and sends emails to the users in order to verify their emails.

RabbitMQ offers horizontal scalability by adding more message brokers which allows it to scale perfectly to a microservice environment. Furthermore, it being open-source means that it is compatible with a lot of technologies and supports all types of messaging protocols. For this application I am using RabbitMQ with the AMQP protocol.

In a highly efficient cluster environment, RabbitMQ shines as a reliable and scalable messaging system. Using Docker, deploying RabbitMQ becomes a swift and seamless process, taking mere seconds to set up. Microservices can effortlessly communicate with RabbitMQ by simply specifying the queue's name and requesting the service by its designated name. This hassle-free integration allows for smooth and efficient interactions within my orchestrated Kubernetes Cluster on Google Cloud platform.

MONGODB DATABASE

One of the strong points when using microservices is that each microservice can use a separate database and still function as a single unit. For the database of choice I chose to use MongoDB. I have contemplated whether to use a relational database like MySQL instead, however I decided against it as when using Typescript I can use the Typegoose Object Data Modeling library to define a reference between different models or classes in MongoBD similar to how they work with relational database.



```
import { getModelForClass, prop, Ref } from '@typegoose/typegoose';
import { User } from './user.model';

export class Session {
  @prop({ ref: () => User })
  user: Ref<User>;

  @prop({ default: true })
  valid: boolean;

  @prop({ default: '' })
  userAgent: string;
}

const SessionModel = getModelForClass(Session, {
  schemaOptions: {
    timestamps: true,
  },
});

export default SessionModel;
```

In this code snippet is an example of how using typescript with typegoose can add some extra functionality to using MongoBD in the form of references between models. This can be used to introduce relationships similar to the relational database one-to-one and one-to-many essentially eliminating most of the advantages that relational databases have over MongoBD and at the same time fully taking advantage of the perks of using a NoSQL database like – not being vulnerable to SQL injections.

CONCLUSION

For this project I am using what is essentially known as a MERN stack or Mongo, Express, React, Nodejs, also known as full-stack JavaScript, which is one of the most popular development platforms and technologies available on the market and there is a reason for that. One of the most obvious advantages is that it requires the knowledge of only one language – JavaScript and it’s derivatives. Furthermore it provides easy scalability and performance because every element of the tech stack is designed to scale horizontally.