Medical application documentation

Student: Razvan Turturica
Group: 30444

Content:

1. Conceptual architecture of the distributed system
2. Database design
3. UML deployment diagram
4. Readme file
5. Asynchronous communication

1. Conceptual architecture of the distributed system

The application should be a web application that will be used for administering medical resources. The application should support Create, Read, Update, Delete operations for various resources. The main resources are the users, there are three types of users: doctors, caregivers, and patients. These three types of users should have different levels of access. Doctors can perform CRUD operations on both caregivers and patients. Caregivers can perform CRUD operations for their assigned patients. Patients can only preview their own medical record. The doctor can assign patients to caregivers as well.

For this requirements I propose an architecture based on two main components: backend and frontend. The backend will be responsible for validating various requests and keeping the state of the application, while the frontend will be an interface that will send requests to the backend and display the application to the user. We will also have a database, for storing the state of the application and will be accessed by the backend. For implementing such an application for web usage, I think that a good technology stack would be the PERN stack. The PERN stands for PostgresSQL, ExpressJS, ReactJ, Node. This stack uses only JavaScript, which can be extended to TypeScript.

The backend will be a RESTful API. This API will listen to requests and resolve them, returning data from the application to the frontend. We will use 4 modules for this: Models, Routes, Controllers, and Middlewares. The Models module will define the resources for the database. In order to access the database, we will use a package named Sequelize that generate the database tables, and will query the database. This file will contain the attributes of each resource and the relationship between them. The Routes module will define the routes that the api will listen to, as well as the Chain of responsibility that uses middleware and controller functions. The Controller module will implement the operations that are required, like Create, Read, Update and Delete. The Middleware module will implement functions to validate the request from the frontend, these functions will make sure there will not be malicious use of the application.

The frontend will implemented in ReactJs using JavaScript. We will have 3 interfaces, one for each role: doctor, caregiver, and patient. All of them will have a signup, login and dashboard page. The doctor dashboard will have a menu to navigate between the DoctorCaregiverDashboard and DoctorPatientDashboard. On these pages there will be a form available for Creating a new user and a table with the current users in the application. This table will have an Actions row that will allow Deletion of the user and Edit. The Edit will open a modal with a precompleted form representing the users data. The caregiver dashboard will be similar with the DoctorPatientDashboard. The patient dashboard will display the the data of the logged user. For implementing this we will use functional components and react hooks. The functional components are similar with component based development, but they are more flexible. We will use packages like react-router-dom to navigate without refresh on the application, semantic-ui-react for displaying html forms with better default styles and more semantic components like Menu and Modal. We will also use styled-components in case we want to easily edit the style on components. We will also have a module to access the API.

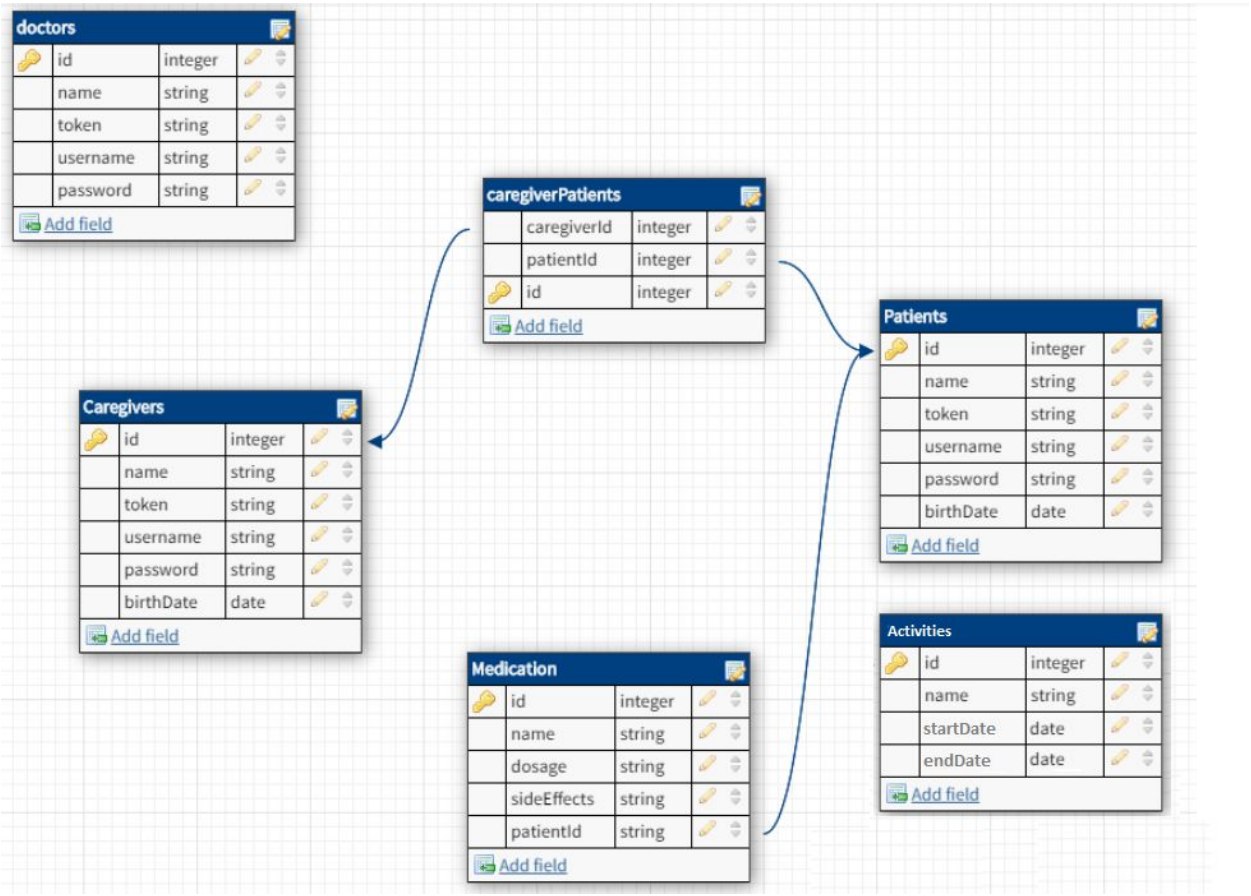
2. Database design

We will use PostgreSQL for database. We will not access it directly, but using a DAL provided by the Sequelize package.

We will use 3 different tables for users, one for each role: doctor, patient, and caregiver. Each of them will have the password encrypted. Each user will only have one authentication token, that meaning that the user will not

be able to be logged from two different places in the same time. The patients will have their medical plan stored in the medications table. The medications are set with a property that deletes them when a patient is deleted. The caregivers and patients are in a many-to-many relationship. A patient can have many caregivers and a caregiver can have many patients. This is done with a Sequelize property that automatically creates the caregiverPatients table and provides additional functions.

The Sequelize package provides a lot of functionalities when it comes to interacting with the database. It can manage additional tables, create hooks, and provide default functions for accessing, saving and interrogating the database. Basically we can interact with the database in an OOP maneer and in JavaScript without the need of writing SQL or sanitizing the data. The models were also described in Sequelize, and they can be used with different databases by simply changing the configuration. For example, the backend tests are run on the same database design, but running on SQLite since it's local and we don't want to deploy another database just for testing.

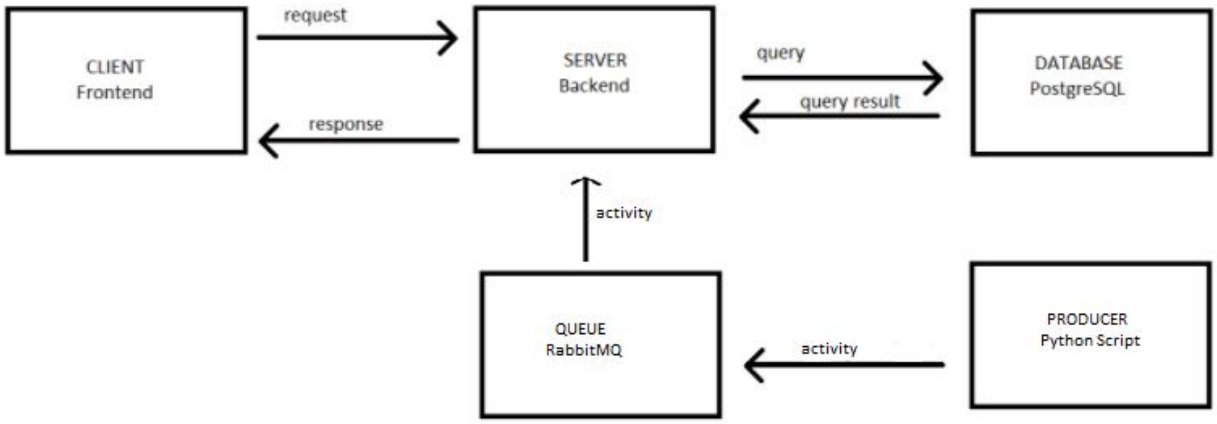


3. UML Deployment Diagram

The website was deployed on Heroku using GitHub. I set up multiple remotes for my repository and both GitHub and GitLab listen to push actions on master. Firstly, I created 2 applications on Heroku and I linked my GitHub with Heroku. After that I wrote a GitHub actions file for the docker image that will execute commands when a push happens on the master branch. The container will build the applications and will deploy them to heroku. The script runs the tests, and if the tests pass, then it will upload the code on the Heroku repository. From there, Heroku will build the application, will compress the slug and will start the application. Also the connection to the database is done with Heroku. It will create a database and will set an environment variable with the connection string.

The application can be found at https://dslab2020-front.herokuapp.com

The backend API is deployed at https://dslab2020-back.herokuapp.com



4. Readme file

Medical application monorepository

This repository contains both backend and frontend of a medical application.

Frontend - React app

Navigate to fronted/

To install the dependencies run `npm install`

To start the development server run `npm start` - the server will start on `localhost:3000`

To run tests run `npm test`

To build production code run `npm build`

Backend - Express app

Navigate to backend/

You need to have postgress installed in order to run the backend, or to set up an SQLite server.

Edit database config edit `config/config.json`

To install the dependencies run `npm install`

To start the development server run `npm start` - the server will start on `localhost:9000`

To run tests run `npm test`

To build production code run `npm build`

5. Asynchronous communication

In this assignment we are required to simulate the existence of a device that is able to report activities of a patient. The device will send information about the activities like the start time, end time and the name of the activity. The doctors and the caregivers will receive real time information about anomalies that happen in patients activities. The rules for these anomalies are sleeping more than 7 hours, staying in the bathroom for more than 30 minutes or going out for more than 5 hours.

To simulate the device I will use a python script that will read data from a file and send them to a server. This script will not send the data directly to the backend. It will use the service RabbitMq in order to collect all the activities in a queue. The backend will read the activities from there and will process them. When an activity is read from the RabbitMq queue it will add it to the database and will check if any anomaly is happening in that activity. If an anomaly is present, the backend will send to the frontend an alert for all the doctors and caregivers to display the anomaly. This communication between the frontend and backend will be through a websocket. To implement it, we can use the socket.io and socket.io-client packages from npm. The frontend will receive the messages and will display them properly.

This design pattern is called the Producer Consumer design pattern and refers to an entity inserting messages in a queue, and the Consumer will remove messages from the queue. In this way we can have multiple entities that communicate asynchronous. The Producers will insert the messages as soon as the events happen, but the Consumers will get them only when they are available. Using a 3rd party service for this is useful when we want to scale applications because the communication overhead will be reduced