**Documentation of Online Medication Platform**

-Distributed systems-

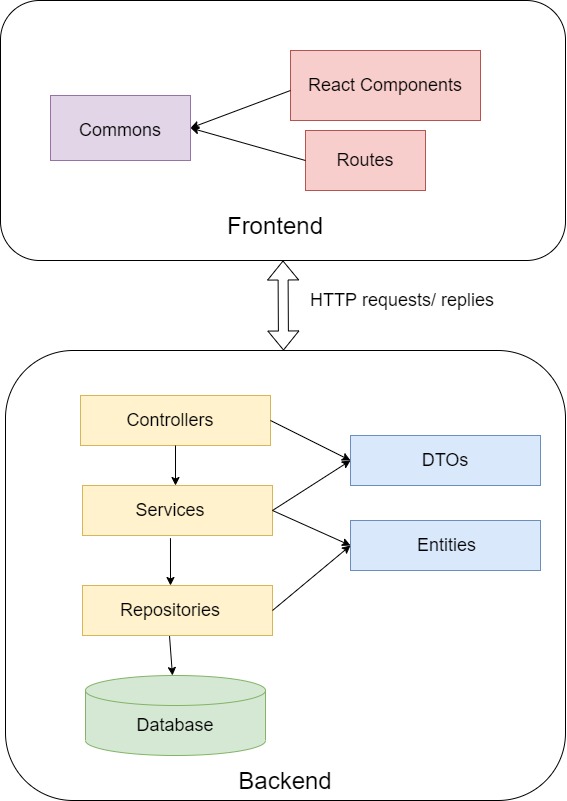
**Name**: Péntek Tamás

**Group**: 30642

This documentation presents a solution for an online medication platform which is designed to manage patients, caregivers and medications. This application has three user types: doctor, which represents the admin role; caregiver and patient. Every type of user has a different webpage, where based on his/her role can do some CRUD operations or can view details about accounts and medications. On the next figure is presented the conceptual architecture of the implemented distributed system. It has three main parts: database, backend and frontend. The communication between frontend and backend is done via REST services, using HTTP methods like POST and GET. Every three main part of this application are deployed on Cloud, but can be tested on localhost too.

In the database are stored data about users and accounts, every necessary information is stored there. The database can be accessed via repository classes, here are implemented some built in queries which are useful to store information in database or to load some information from database. An entity is a model for a table from the relational database and each object with type entity corresponds to a row from the database. DTO (Data Transfer Objects) are some special object types which are transferred outside the backend application to the frontend application. The objects are used to transfer data more safely between backend and frontend, but in the database are store only entity data, so every time when an object arrives from fronted, it is converted into an entity object and every object sent by backend to frontend is converted from entity to DTO object. The service layer represents the business logic of the backend application, here are connected data from frontend with the data from backend, converting from DTO to entity and from entity to DTO. To make this conversion much easier, for every entity type is declared a Builder class which makes the conversion between different data types. The Controller layer is the closest layer to frontend, here are linked the functionalities of other classes, practically controllers represents an API for handling HTTP request from frontend.

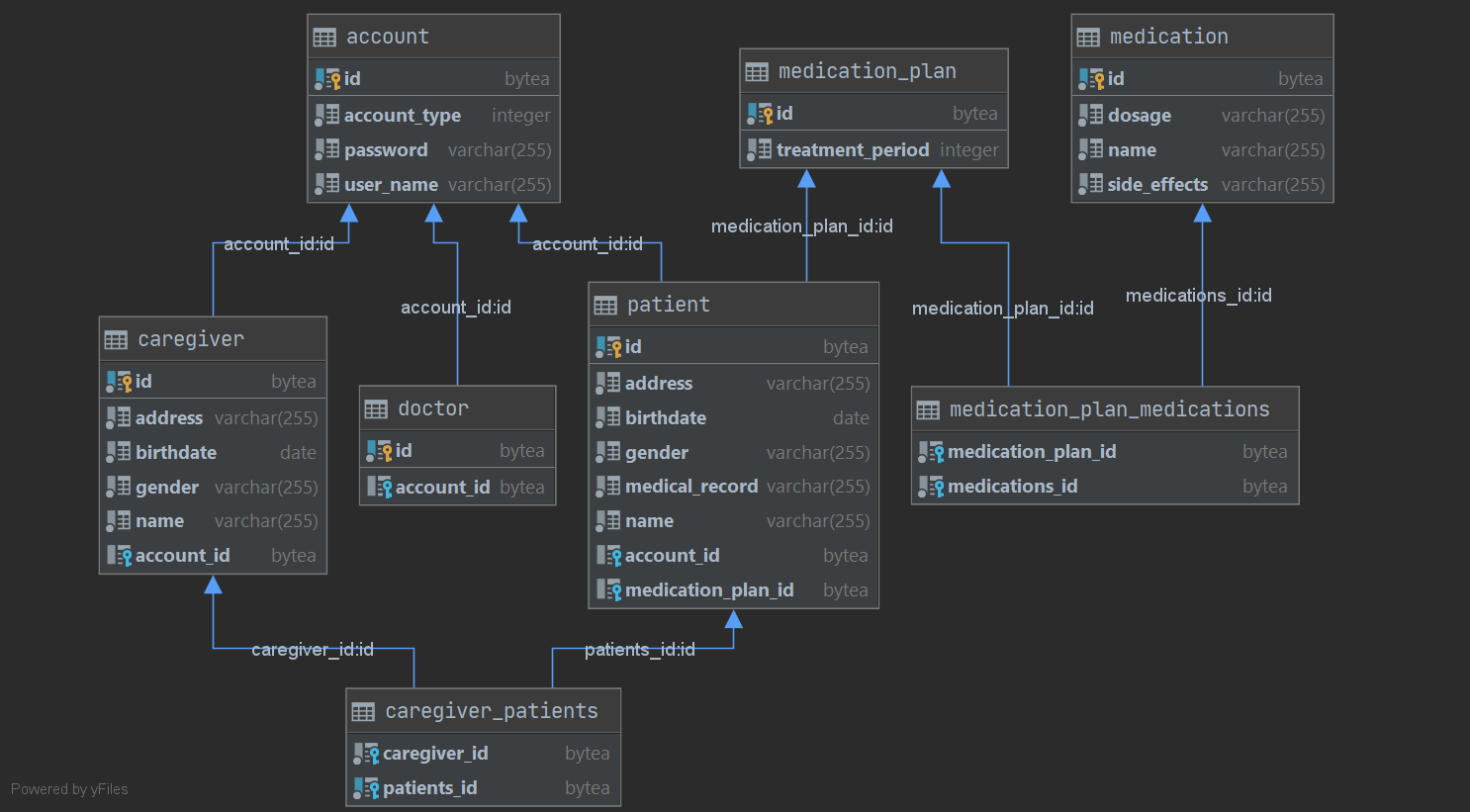
The frontend part of the application has also three main parts: Commons, Components and Routes. Commons contains a set of modules, for each route exists a module and each module includes some functionalities for communication or error handling purposes. Routes are used to navigate between different pages, to access different data and functionality. Components are used to visualize buttons, tables, lists. The main purpose of a Component is to offer a user-friendly interface for a better interaction with the application.



**Figure 1. Conceptual description of the distributed system**

On the next figure is presented the diagram of the relational database. This application has six main tables: account, caregiver, doctor, patient, medication and medication\_plan. In account table are stored the user credentials user for authentication in case of each type of user. Near this each type of user has a separate table, in these tables are stored personal data about each user. Medication table contains data about available medicaments for patients. In medication\_plan can be stored a plan made by a doctor for a patient.

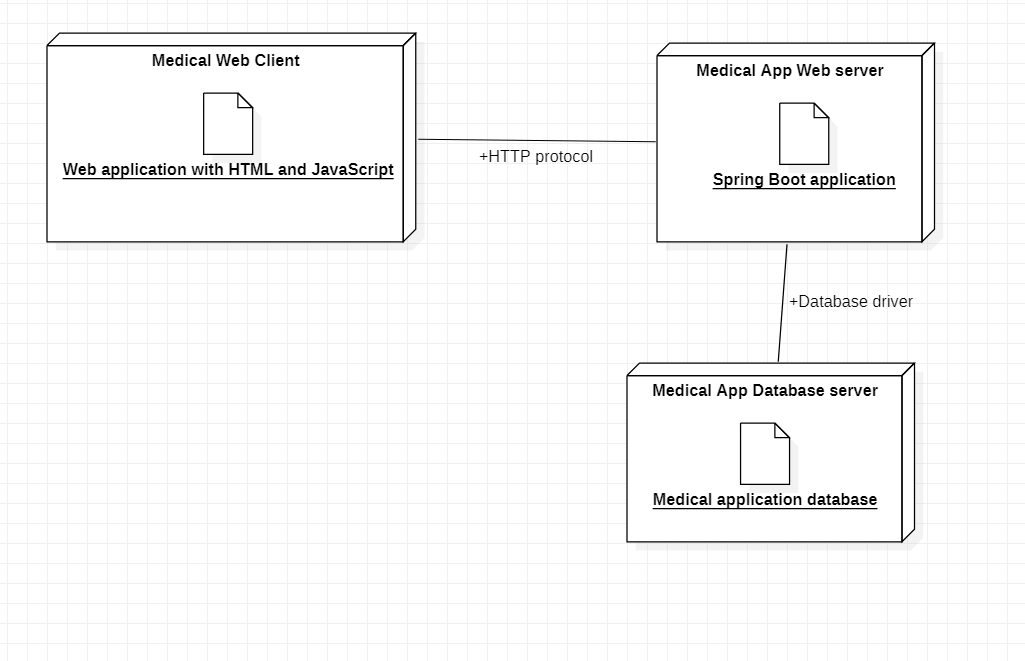
The others tables are automatically generated by Hibernate, based on the entity classes. These tables are useful to make relation between different tables. As Figure 2 shows, between tables we have different type of relations, like one-to-one between doctor table and account table, but near this we can find one-to-many or many-tom-may relations too.



**Figure 2. Relationa database design**

On the next diagram is presented the Deployment diagram of the distributed system. The first part of this diagram is a relational database server, on which is stored the database of the application. In this case the database server is Postgres, which is an open source database server and it has an add-on for Heroku Cloud, so it is useful when the application is planned to be deployed on cloud. The second part is represented by the web server of the system, which is implemented using Spring Boot. This framework generates all the configurations used by the server and in this way the developer can concentrate on more important things. The communication between database server and web server is done by a database driver, which offers an API for a better communication. On the other side we have web client implemented in React, which generates JavaScript and HTML files which are used to display data in a web browser. In the frontend part of the application are created a lot of React Components to have a better visualization of the data from the relational database. The communication between backend and frontend is done by HTTP protocol using mainly GET and POST methods to transfer data in both directions.

The frontend is built using node and the backend using maven. This distributed system is deployed on Heroku cloud using ruby gems. The application can be tested locally, using a local database, for example MySQL or Postgres and both frontend and backend running on localhost. Near this, the application can be accessed from cloud, which is a much better solution, because this application can be accessed from everywhere, without location restrictions. This application is easy to use, the main page is the login page, where each user can enter his/her credentials and after that is redirected to a page based on his/her role. The admin of the application is represented by the doctor user type, he/she can view every information about other users and about medications and medication plans. There is no possibility to create a doctor user via user interface, so it can be done by inserting direct in the database or by using a tool like Postman, where we can make a POST method with a JSON, which contains details about doctor account.



**Figure 3. UML Deployment diagram**