DISTRIBUTED SYSTEMS
ASSIGNMENT 3

WEB SOCKETS AND SECURITY

Student: Deac Dan Cristian

Faculty of Automation and Computer science

Group: 30443

CONTENTS

SPECIFICATIONS

This application represents an Energy Management System that comprises a frontend and multiple microservices created to oversee users and their linked smart energy measuring devices. After a login process, the system can be accessed by two categories of users: supervisor (admin), and customers. The supervisor can execute CRUD (Create-Read-Update-Delete) actions on user accounts (defined by ID, name, role: admin/client), smart energy measuring devices (defined by ID, description, location, maximum hourly energy utilization), and on the linking of users to devices (each user can possess one or more smart devices in various places).

Each device is supposed to be monitored and send consumption data to the application. This is done using a simulation script. The information is then sent through RabbitMQ to a receiver service. The Receiver microservice is getting data from a queue hosted on a RabbitMQ container where sensor data is sent from a transmitter. Then this data can be queried by the front end to show user specific data, or notifications can be sent to notify the user in case of excessive readings from their devices' sensors.

The security of the application is ensured by a JWT spring security authentication mechanism. When a user logs in, their token is given by the User microservice who generates it, and then stored in local storage for a set amount of time, and when a user wants to do an action calling a request, their token gets verified in the respective microservices for a valid token.

The users can chat with an administrator when something goes wrong. This is done by the Chat microservice using WebSockets to send messages back and forth, on a message topic.

KEY CHARACTERISTICS:

Assignment 1:

- User login is required. Users are redirected to a suitable page according to their role.
- Admin role
  - CRUD operations on users
  - CRUD operations on devices
  - Creating user-device associations
- User/Client role
  - Can view their page with all linked devices.
- Users in one role are prohibited from accessing the pages of other roles (e.g., by logging in and then copying the admin URL in the browser)

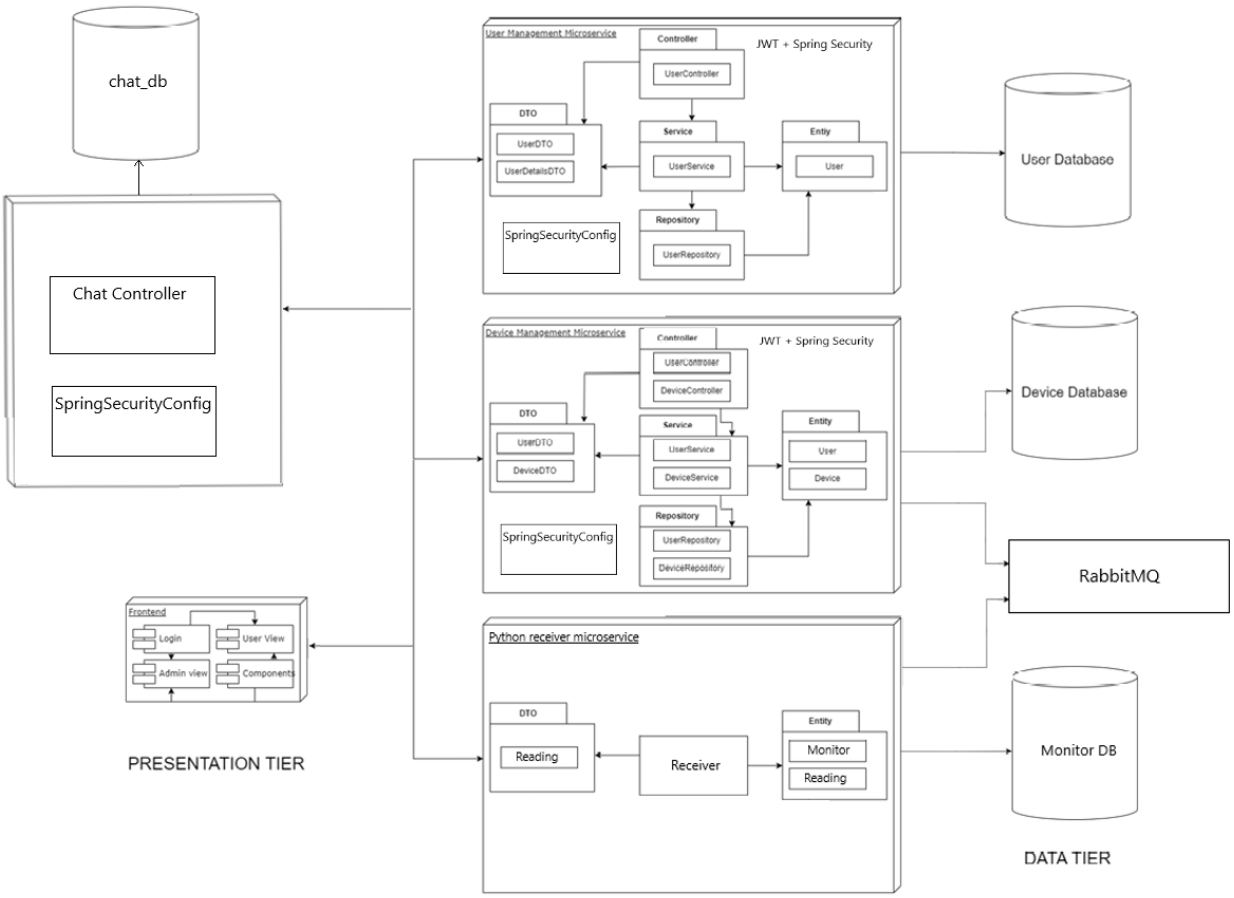
Assignment 2:

- The message-oriented middleware allows the Smart Metering Device Simulator to send data tuples in a JSON format.
- The message consumer component of the microservice processes each message and notifies asynchronously the client application using WebSocket.
- The hourly energy values will be saved by the consumer component in the Monitoring database.

Assignment 3:

- Authentication is finally available fully with spring security and jwt. We generate tokens when a user authenticates and use those tokens in the requests the frontend sends.
- The token gets stored in local storage.
- Additionally, using the token information, we can implement the Chat functionality, allowing users to chat with admins. An admin can chat with multiple users, and there is also a notification for the "seen" functionality, implemented using build int react functions.

ARCHITECTURE



This application consists of a React front-end and a few distinct microservices implemented in Spring REST and a python microservice with FLASK. These microservices manage different aspects of the application: User Management, Device Management, a chat, and Sensor Reading. The architecture employs separate databases for each microservice, emphasizing the organization of the User Management microservice with a 'user_db' database housing the 'users' table, as well as a authenticator and token generator, and the Device Management microservice functioning with a 'devices_db' database containing 'devices' and 'users' tables and the validation logic for tokens. In the python microservice we store the readings in a Monitor table, and the hourly measurements are sent to the Measurements table, called “Readings” in this diagram. In the chat microservice, we host the websocket commhnication logic.

FRONT-END: REACT + JAVASCRIPT

The front-end of the application is developed using React, a modern JavaScript framework recognized for its simplicity and adaptability. React allows for the development of interactive and user-friendly interfaces, enabling smooth interactions between users and the application. Its component-based structure simplifies the development process and improves maintainability.

BACK-END MICROSERVICES

USER MANAGEMENT MICROSERVICE

The User Management microservice is a foundational component of the application. It manages all user-related functionalities. This microservice is built using Spring REST, a robust and versatile framework for constructing scalable web services.

DATABASE:

- user_db

TABLES:

- users:

DEVICE MANAGEMENT MICROSERVICE

The Device Management microservice is responsible for overseeing devices within the application. It is also developed using Spring REST.

DATABASE NAME:

- devices_db

TABLES:

- device: Stores device-specific information.
- users: Contains a replicated version of the 'users' table from the User Management microservice to support device-related functionalities that require user data.

The decision to replicate the 'users' table in the Device Management microservice's database aims to enhance performance and reduce dependencies between microservices. It enables the Device Management microservice to efficiently handle device operations without continuous communication with the User Management microservice.

RECEIVER MICROSERVICE(PYTHON)

The Receiver microservice is responsible for reading sensor data sent to a RabbitMQ Queue and providing the data for the front end to display it to the users.

DATABASE NAME:

- Monitor_db

TABLES:

- Monitor: Stores device readings.
- Measurements: Stores all the hourly data for each device.

CHAT MICROSERVICE

This service allows for chat functionality, allowing users to talk with admins to get help on different issues.

DATABASE:

- chat_db

TABLES:

- messages:

COMMUNICATION BETWEEN MICROSERVICES

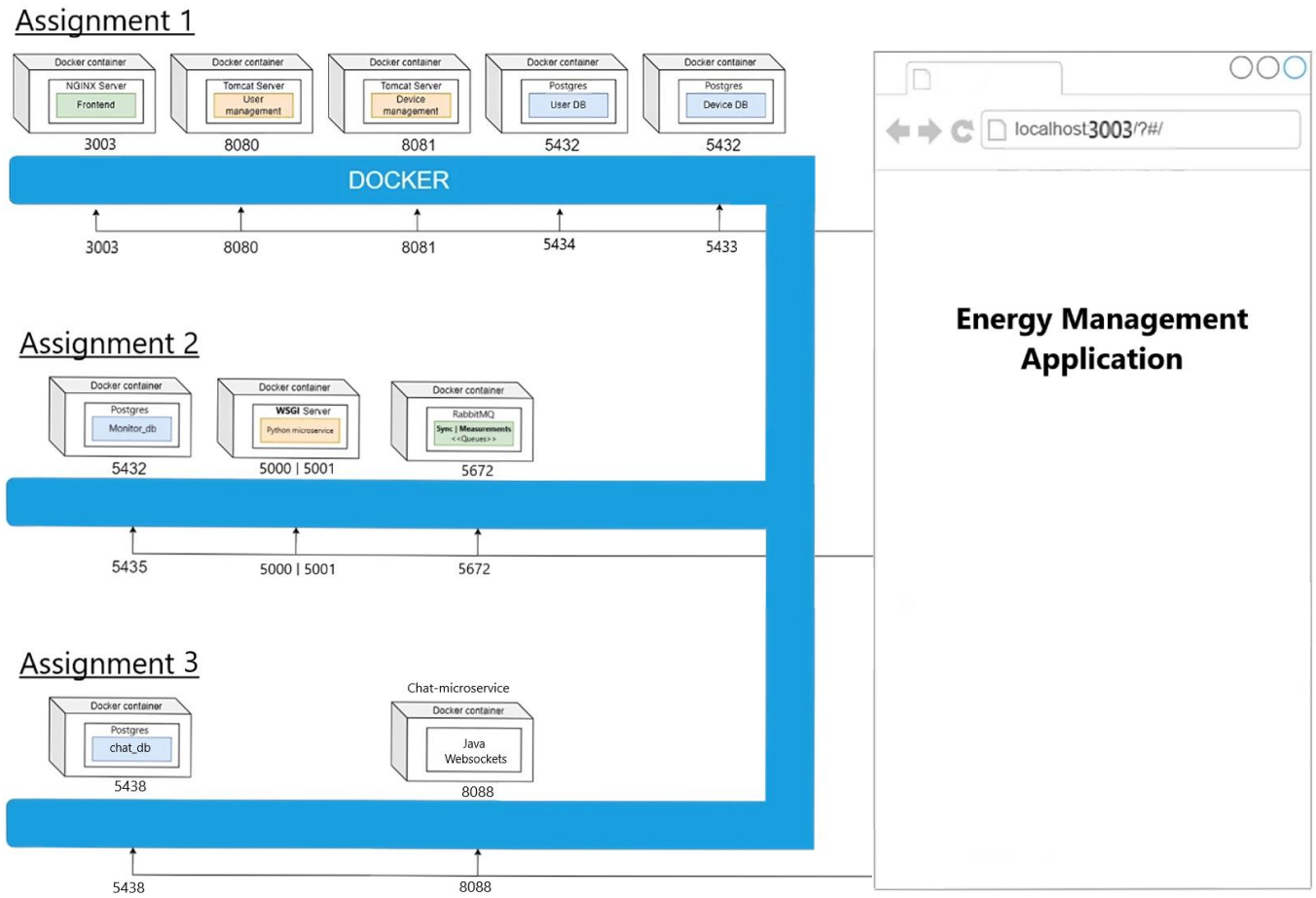
Microservices communicate with each other through well-defined RESTful APIs or FLASK in case of the Python microservice, facilitating a loosely coupled and modular structure. Communication primarily occurs through HTTP requests, ensuring interoperability and enabling each microservice to operate independently while fulfilling its designated functionality, but makes use of Web sockets and RabbitMQ queues.

Web sockets are implemented in the chat microservice, bringing chat functionality to the application. It allows an admin to chat with multiple users using websockets.

The architecture of the full-stack application guarantees an organized and efficient approach to managing users and devices. With React on the front-end, the application offers an intuitive user interface. Meanwhile, the Spring REST-based microservices and their corresponding databases facilitate the efficient handling of user and device-related operations, creating a scalable and modular system that encourages flexibility and maintainability. The division of responsibilities between the multiple microservices, coupled with the replication of user data in the Device Management microservice, synchronizing the databases, optimizes performance and enhances the overall functionality of the application.

DEPLOYMENT IN DOCKER

The deployment strategy for the full-stack application involves Docker, a containerization platform that enables the encapsulation of each application component into separate containers. This chapter details the deployment configuration for the React front-end, User Management microservice, Device Management microservice, Receiver microservice, Chat microservice, and their respective databases, which are isolated within individual Docker containers. These containers expose internal ports to enable interaction with external systems, such as a web browser for the front-end.



CONTAINERIZATION CONFIGURATION

FRONT-END CONTAINER (REACT)

- The React front-end application is enclosed in a Docker container. The container encompasses all essential front-end resources, including HTML, CSS, JavaScript, and any other static assets. It utilizes NGINX as the web server.
- Docker Container Name: frontend
- Internal Docker Port: 3000
- Exposed Port: 3003
- The internal port 3000 within the Docker container is made accessible externally on port 3003, allowing interaction with the application through a web browser.

USER MANAGEMENT MICROSERVICE CONTAINER

- The User Management microservice, constructed using Spring REST, is containerized for deployment. This container has the role of the token generator service. When a user authenticates, a token is generated used for validating other requests to other services who have implemented the validation logic.
- Docker Container Name: users-microservice
- Internal Docker Port: 8080
- Exposed Port: 8080
- The internal port 8081 of the User Management microservice container is exposed directly, facilitating communication with other components and external systems.

DEVICE MANAGEMENT MICROSERVICE CONTAINER

- Like the User Management microservice, the Device Management microservice is also containerized.
- Docker Container Name: devices-microservice
- Internal Docker Port: 8081
- Exposed Port: 8081
- The internal port 8081 of the Device Management microservice container is exposed for interaction and integration with other system components.

RECEIVER MICROSERVICE CONTAINER

- Receiver microservice is also containerized.
- Docker Container Name: python-microservice
- Internal Docker Port: 5000 and 5001
- Exposed Port: 5000 and 5001 respectively
- This service has multiple running threads, one of them using port 5000 and the other 5001 for their specific tasks.

CHAT MICROSERVICE CONTAINER

- The chat microservice, constructed using Spring REST, is containerized for deployment.
- Docker Container Name: chat-microservice
- Internal Docker Port: 8088
- Exposed Port: 8088
- The internal port 8088 of the Chat microservice container is exposed directly, facilitating communication with other components and external systems.

USER DATABASE CONTAINER

- The User Management microservice's database, 'user_db,' is contained within its designated Docker container.
- Docker Container Name: postgres
- Internal Docker Port: 5432
- Exposed Port: 5434
- The internal port 5432, where the user database operates, is exposed to the external system for accessing user-related data.

CHAT DATABASE CONTAINER

- The Chat microservice's database, ‘chat_db,' is contained within its designated Docker container.
- Docker Container Name: postgres-chat
- Internal Docker Port: 5432
- Exposed Port: 5438
- The internal port 5432, where the chat database operates, is exposed to the external system for accessing user-related data.

DEVICE DATABASE CONTAINER

- The database for the Device Management microservice, 'devices_db,' is housed within its respective Docker container.
- Docker Container Name: postgres
- Internal Docker Port: 5432
- Exposed Port: 5433
- The internal port 5432, dedicated to device database operations, is exposed to facilitate communication and data management for devices.

MONITOR DATABASE CONTAINER

- The database for the Receiver microservice, 'Monitor_db,' is housed within its respective Docker container.
- Docker Container Name: postgres
- Internal Docker Port: 5432
- Exposed Port: 5435
- The internal port 5432, dedicated to device database operations, is exposed to facilitate communication and data management for devices.

DOCKER SUMMARY

The exposed ports of the Docker containers enable interaction with the application components through a standard web browser. Users can access the front-end application hosted on the exposed port 3003, which then communicates with the microservices through their respective exposed ports (8080, 8081, 8088, 5001) to perform related operations. All containers are run from the same docker-compose.yml file present in the root folder of the repository.

CONCLUSION

The deployment of the application components within Docker containers ensures encapsulation and isolation, enhancing portability, scalability, and consistency across various environments. Each component, including the front-end, microservices, and databases, is housed within its designated container, facilitating interaction via exposed ports for seamless communication and operation. The configuration allows external systems, such as web browsers, to interact with the application's front-end, initiating requests to the user and device microservices through their exposed ports, thereby enabling a fully functional and integrated system. This project has enabled me to understand more about how real world applications are deployed and has improved my developer portfolio significantly.