**User Interface Design**

Project Documentation

Student : Lazăr Cătălin-Mihai

Group: 30443

**Client microservice**

This microservice contains the logic for the client-related operations. It has 5 main packages: controllers, dtos, entities, repositories and services.

***Controllers:***

This is the layer that exposes the application functionality as an API able to handle HTTP REST requests. It also contains handlers for various types of exceptions. It contains different endpoints for getting all the clients from the database, inserting a client, deleting a client and updating a client.

Example of an endpoint for deleting a client:

@DeleteMapping(value = "/{id}")  
public ResponseEntity<Void> deleteClient(@PathVariable("id") Integer clientId){  
 boolean isRemoved = clientService.deleteClientById(clientId);  
 if(!isRemoved){  
 return new ResponseEntity<>(HttpStatus.*NOT\_FOUND*);  
 }  
 return new ResponseEntity<>(HttpStatus.*NO\_CONTENT*);  
}

***Services:***

This layer represents the business logic layer of the Spring application. It translates the Data Transfer Objects (DTOs) into entities and back. For formatting the values from DTO objects to Entity objects Builder classes are used. The service layer is responsible to apply more complex operations and validations before accessing the repository layer.

The deleteClientById function that is called from the Controller:

public boolean deleteClientById(Integer clientId) {  
 if(clientRepository.existsById(clientId)){  
 clientRepository.deleteById(clientId);  
 *LOGGER*.debug("Client with id {} was deleted from client-db", clientId);  
 return true;  
 } else { *LOGGER*.debug("There is NO client with id {} in client-db", clientId); return false; }}

In the insert function I also have the functionality to let the device microservice know when a user is added, so that it is also added in the device microservice database. This is done using RestTemplate.

***DTOs:***

A Data Transfer Object (DTO) is a special object exposed outside the application (to the UI or APIs). It contains part of the underlying Entities or combinations of different entities. Additionally, it contains builders and validators.

There are two DTO classes, ClientDetailsDTO and ClientDTO. The difference is that ClientDTO doesn’t contain all the details, and is used when retrieving information from the database to the frontend. For example we might not want to show the password of a client when retrieving one.

The field of ClientDetailsDTO:

private Integer id;  
@NotNull  
private String name;  
  
@Nullable  
private String password;  
@NotNull  
private String role;

The fields of ClientDTO:

private Integer id;  
private String name;

***Repositories:***

Package that contains the repositories, classes that facilitate the DB access. The developer can use custom queries to communicate with the DB .

***Entities:***

An entity corresponds to a table from the relational database and each instance of the entity corresponds to a row from the database. The Client entity contains a name, a password and a role (client/admin).

**Device microservice**

The Device microservice contains the same structure as the Client microservice. One difference worth noting is that the Device microservice also contains in the Entities package an entity for clients. This is because the Device database contains a table for clients and also a table for the devices that each client has.

**Frontend**

**Api package**

The api package contains helper files for managing API requests to backend services.

Example:

userApi.js: Contains the link for user-related API endpoints, localhost:8080

deviceApi.js: Contains the link for user-related API endpoints, localhost:8081

***Components package:***

The components folder holds reusable UI components used across different parts of the app. Components here are typically smaller pieces of the UI that are not tied to any specific page and can be reused in multiple locations, such as buttons, form fields, modals, and navigation elements.

Example:

Navbar.js: The main navigation bar component, allowing users to switch between different sections of the app (e.g., User Management and Device Management).

UserForm.js: A form component used for creating and editing user information, handling fields like name, role, and password.

DeviceForm.js: A form component used for managing device data, such as adding or editing device details.

***Pages package:***

The pages folder contains the main page components, each corresponding to a different route or feature in the application. Each page typically combines smaller components and API calls to create a complete view, handling the core functionality of the app, such as displaying data and performing CRUD operations.

Example:

UserManagement.js: A page that displays the list of users, along with options to add, edit, and delete users. It leverages UserForm and calls userApi.js functions for API interactions. DeviceManagement.js: A page for managing devices, displaying the list of devices with options to add, edit, and delete device records. It integrates DeviceForm and calls deviceApi.js functions to handle device-related data.

**Assignment 2**

**Monitoring microservice**

This microservice connects to a RabbitMQ queue hosted on CloudAMQP and takes the data send from the python script that represent the readings for a device and prints these readings on the screen.

The main classes of this microservice are:

***RabbitMQConfig***

This class declares the queue and its properties.

@Bean  
public Queue queue() {  
 return new Queue(queueName, true, false, false); *// Ensure durable queue*}

***MessageConsumer***

This is the class that receives the messages from the CloudAMQP queue, transforms them into a JSON object and prints them on the screen.

@RabbitListener(queues = "${rabbitmq.queue-name}")public void receiveMessage(String message) {  
 ObjectMapper mapper = new ObjectMapper();  
 try {  
 JsonNode jsonNode = mapper.readTree(message);  
 System.*out*.println("Parsed Message: " + jsonNode.toString());  
 } catch (JsonProcessingException e) {  
 e.printStackTrace();  
 }  
}

***Simulator script***

This script reads sensor data from a sensor.csv file and sends it to a RabbitMQ queue. First the script loads configuration details from a config.json file and then establishes a RabbitMQ connection and declares the queue.

The script extracts numeric sensor values from the CSV file and puts them with a UTC timestamp and a device ID into a JSON message, which is then sent to the queue.

This is the connection to the queue:

# Connect to RabbitMQ

connection\_params = pika.URLParameters(rabbitmq\_url)

connection = pika.BlockingConnection(connection\_params)

channel = connection.channel()

channel.queue\_declare(queue=queue\_name, durable=True)

**Assignment 3**

**Chat Microservice**

***WebSocketConfig***

It configures WebSocket messaging in the application and enables a message broker for broadcasting messages to destinations like /topic. It also sets the application prefix for WebSocket destinations as /app.

The function registerStompEndpoints registers the /chat WebSocket endpoint, allowing connections from all origins (\*).

@Override  
public void configureMessageBroker(MessageBrokerRegistry registry) {  
 registry.enableSimpleBroker("/topic"); *// Broadcast destination* registry.setApplicationDestinationPrefixes("/app");  
}  
  
@Override  
public void registerStompEndpoints(StompEndpointRegistry registry) {  
 registry.addEndpoint("/chat")  
 .setAllowedOrigins("\*");  
}

***SecurityConfig***

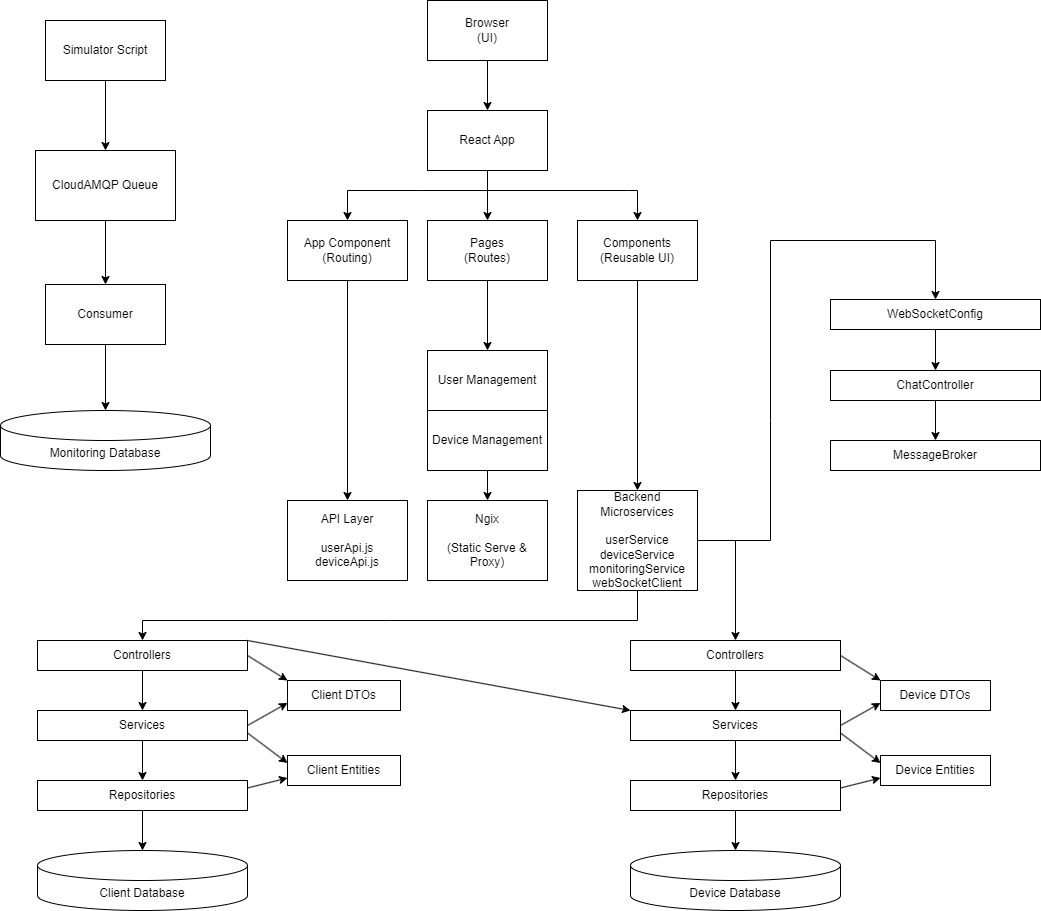
This class configures security settings for the application. It Disables CSRF protection to simplify WebSocket communication and permits unauthenticated access to WebSocket-related endpoints (/chat, /app/\*\*, /topic/\*\*) while securing other routes.

@Bean  
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
 http  
 .csrf().disable()  
 .authorizeHttpRequests((authz) -> authz  
 .requestMatchers(new AntPathRequestMatcher("/chat"),  
 new AntPathRequestMatcher("/app/\*\*"),  
 new AntPathRequestMatcher("/topic/\*\*")).permitAll()  
 .anyRequest().authenticated()  
 )  
 .headers().frameOptions().disable();  
 return http.build();  
}

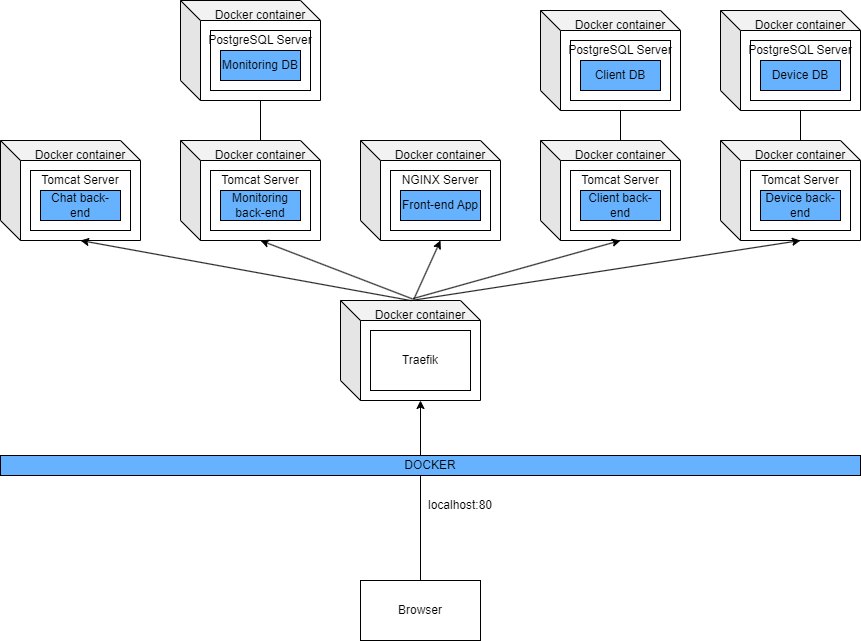
***ChatController***

It handles WebSocket messages sent to the /app/send endpoint and broadcasts incoming chat messages to all subscribers of the /topic/messages destination.

@MessageMapping("/send")  
@SendTo("/topic/messages")  
public ChatMessage handleMessage(ChatMessage message) {  
 return message;  
}

****

Conceptual Architecture diagram



Deployment diagram