***PRACTICA DE VARA***

***2024***

***LIFT APPLICATION***

***BACK END***

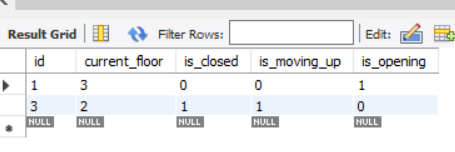
***STUDENT : YLHAM MYRAOV Prof.DR.ING MARIA MAGDELENA SANTA***

***GRUPA 30133 Prof.DR.ING PETRU DOBRA***

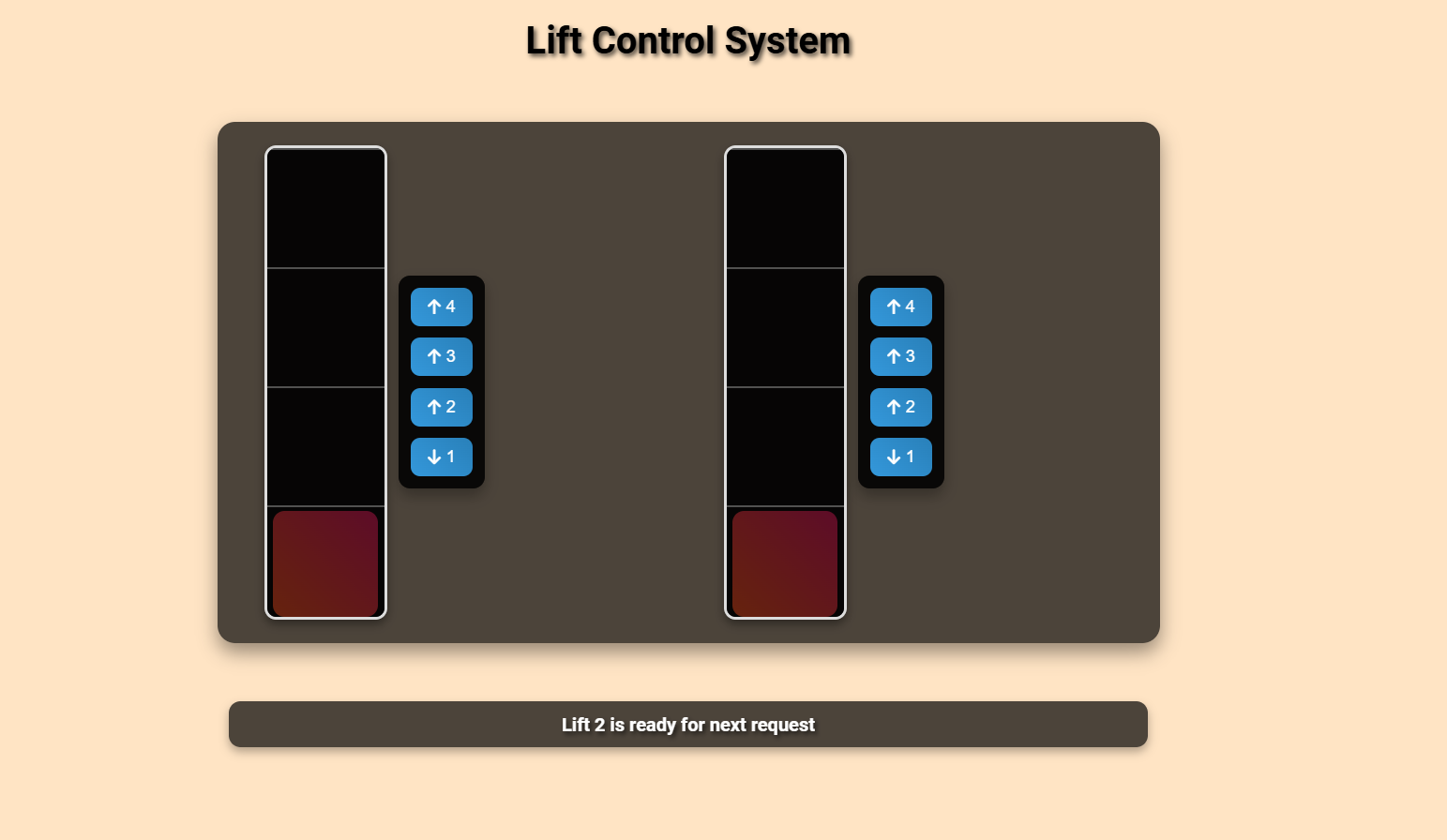
# Dezvoltarea Aplicației de Control al Liftului Backend

## Introducere

În cadrul acestui proiect, am dezvoltat o aplicație de control al liftului folosind Spring Boot, Java, MySQL. Obiectivul principal a fost crearea unei aplicații complete care să gestioneze cererile utilizatorilor, să inițieze mișcarea liftului și să monitorizeze sosirile la diferite etaje ale clădirii.



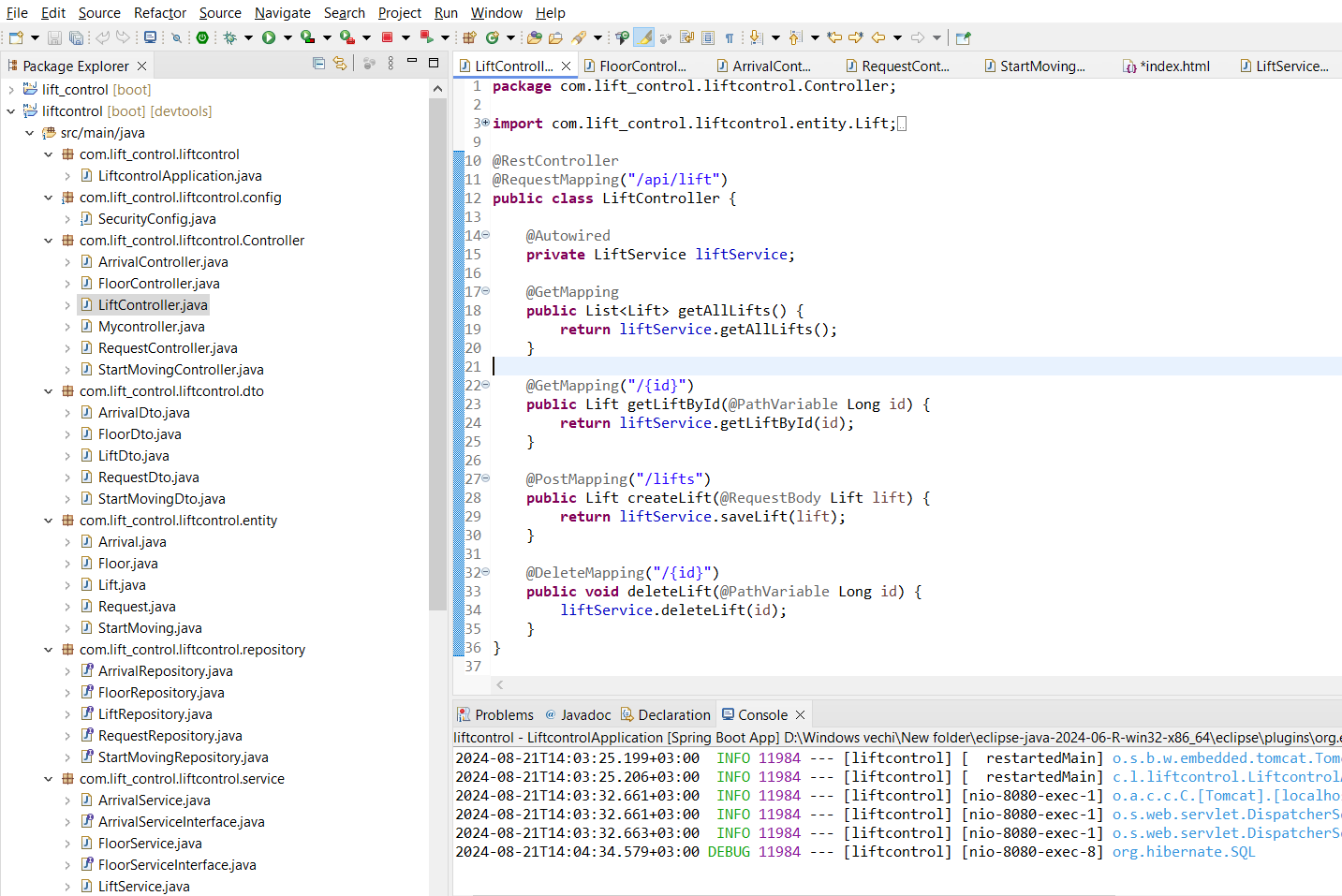
Frontend



## Etapele Dezvoltării

### 1. Planificarea și Designul

**Planificarea:**

* Am identificat principalele componente și funcționalități necesare pentru aplicație, cum ar fi gestionarea cererilor, inițierea mișcării liftului, monitorizarea sosirilor și gestionarea etajelor.
* Am realizat o diagramă a componentelor pentru a înțelege cum vor interacționa diferitele părți ale aplicației.
* 

**Designul:**

* Am creat o diagramă de clasă pentru a modela entitățile din baza de date, inclusiv Lift, Request, StartMoving, Arrival și Floor.
* Am utilizat rețele Petri pentru a modela comportamentul sistemului și a vizualiza fluxul de lucru între diferitele componente.

### 2. Setarea Mediului de Dezvoltare

**Instrumente utilizate:**

* **Eclipse:** IDE pentru dezvoltarea în Java.
* **MySQL:** Sistem de management al bazelor de date relaționale pentru stocarea datelor.
* **Spring Boot:** Framework pentru dezvoltarea backend-ului.
* **Frontend:** HTML, CSS și JavaScript pentru crearea interfeței utilizatorului.

**Configurarea proiectului:**

* Am creat un nou proiect Spring Boot folosind Spring Initializr.
* Am adăugat dependențele necesare în fișierul pom.xml (Spring Web, Spring Data JPA, MySQL Driver).
* Am configurat conexiunea la baza de date în fișierul application.properties.

### 3. Implementarea Backend-ului

**Entități:**

* Am definit clasele entitate (Lift, Request, StartMoving, Arrival, Floor) folosind anotarea @Entity, care marchează aceste clase ca fiind modele pentru tabelele din baza de date.

**Repository-uri:**

* Am creat repository-uri pentru fiecare entitate folosind Spring Data JPA (LiftRepository, RequestRepository, StartMovingRepository, ArrivalRepository, FloorRepository), care se ocupă cu operațiile CRUD (Create, Read, Update, Delete) asupra datelor.

**Servicii:**

* Am implementat servicii (LiftService, RequestService, StartMovingService, ArrivalService, FloorService) care conțin logica de afaceri. Aceste servicii interacționează cu repository-urile pentru a gestiona datele și a executa operațiuni complexe.

**Controller:**

* Am creat un controller REST (LiftController) pentru a gestiona cererile HTTP și a expune endpoint-urile API. Acest controller primește cererile utilizatorilor și le trimite către serviciile corespunzătoare.

***SecurityConfig***

**Ce este SecurityConfig**:

* **Rolul său**: Clasa SecurityConfig este responsabilă pentru configurarea securității în aplicația ta. Aici definești reguli pentru autentificare (cum se loghează utilizatorii) și autorizare (ce resurse pot accesa diferite roluri de utilizatori).
* **De ce e importantă**: Asigură că aplicația ta este protejată împotriva accesului neautorizat și a altor riscuri de securitate. De exemplu, poți bloca accesul la anumite pagini sau funcționalități pentru utilizatorii care nu au roluri adecvate
* package com.lift\_control.liftcontrol.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.core.userdetails.User;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.provisioning.InMemoryUserDetailsManager;

import org.springframework.security.web.SecurityFilterChain;

* @Configuration
* @EnableWebSecurity
* public class SecurityConfig {
* @Bean
* public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
* http
* .csrf(csrf -> csrf.disable())
* .authorizeHttpRequests(auth -> auth
* .requestMatchers("/api/\*\*").authenticated()
* .anyRequest().permitAll()
* )
* .httpBasic(httpBasic -> {});
* return http.build();
* }
* @Bean
* public UserDetailsService userDetailsService() {
* UserDetails apiUser = User.builder()
* .username("apiuser")
* .password(passwordEncoder().encode("liftapi123"))
* .roles("API")
* .build();
* return new InMemoryUserDetailsManager(apiUser);
* }
* @Bean
* public PasswordEncoder passwordEncoder() {
* return new BCryptPasswordEncoder();
* }
* }

**1. Diagrama de Componente cu Porturi și Interfețe**

**Diagrama de Componente** ilustrează structura aplicației tale și modul în care componentele sale interacționează între ele. În cazul aplicației de control al liftului, avem următoarele componente:

1. **LiftControlApplication**: Clasa principală care pornește aplicația Spring Boot.
2. **LiftController**: Gestionează cererile HTTP și le trimite către serviciile corespunzătoare.
3. **LiftService**: Conține logica de afaceri pentru gestionarea lifturilor.
4. **LiftRepository**: Interacționează cu baza de date pentru a salva și a prelua informații despre lifturi.
5. **Lift**: Entitatea care reprezintă datele unui lift.
6. **RequestService**: Gestionează cererile de mișcare ale liftului.
7. **RequestRepository**: Interacționează cu baza de date pentru a salva și a prelua cererile.
8. **Request**: Entitatea care reprezintă cererile utilizatorilor.
9. **StartMovingService**: Gestionează inițierea mișcării liftului.
10. **StartMovingRepository**: Interacționează cu baza de date pentru a salva și a prelua informații despre mișcare.
11. **StartMoving**: Entitatea care reprezintă inițierea mișcării liftului.
12. **ArrivalService**: Gestionează sosirile liftului la etaje.
13. **ArrivalRepository**: Interacționează cu baza de date pentru a salva și a prelua informații despre sosiri.
14. **Arrival**: Entitatea care reprezintă sosirile liftului.
15. **FloorService**: Gestionează informațiile despre etaje.
16. **FloorRepository**: Interacționează cu baza de date pentru a salva și a prelua informații despre etaje.
17. **Floor**: Entitatea care reprezintă etajele clădirii.

**Interacțiuni**:

* **LiftController** comunică cu toate serviciile (LiftService, RequestService, etc.) pentru a procesa cererile.
* **LiftService**, **RequestService**, **StartMovingService**, **ArrivalService**, și **FloorService** interacționează cu repository-urile respective pentru a accesa datele.
* **Repository-uri** (LiftRepository, RequestRepository, etc.) sunt responsabile pentru interacțiunea cu baza de date.

**Exemplu de Diagramă**:

[ LiftControlApplication (main) ]

|

v

[ LiftController ] <--> [ LiftService ] <--> [ LiftRepository ] <--> [ Lift ]

| |

v v

[ RequestService ] <--> [ RequestRepository ] <--> [ Request ]

|

v

[ StartMovingService ] <--> [ StartMovingRepository ] <--> [ StartMoving ]

|

v

[ ArrivalService ] <--> [ ArrivalRepository ] <--> [ Arrival ]

|

v

[ FloorService ] <--> [ FloorRepository ] <--> [ Floor ]

v

[ Securityconfig ] <--> [ dto ] <--> [ floor request lift arrival startmoving ]

v

[ Implement services ] <--> [ all classes ]

[javascript] <--> [html] <--> [ css ]

**2. Rețele Petri Integrate în Componente**

Rețelele Petri sunt un instrument pentru modelarea și analiza fluxurilor de lucru. Ele ajută la vizualizarea stărilor și tranzițiilor dintre acestea.

**Locații**:

1. **CererePrimita**: Starea în care cererea de mișcare a fost primită (clasa Request).
2. **ProcesareCerere**: Starea în care cererea este procesată (clasa LiftService).
3. **LiftInMiscare**: Starea în care liftul este în mișcare (clasa StartMoving).
4. **AjunsLaDestinatie**: Starea în care liftul a ajuns la destinație (clasa Arrival).
5. **EtajCurent**: Starea în care etajul curent este actualizat (clasa Floor).

**Tranziții**:

1. **PrimireCerere**: Metoda createRequest din RequestService transformă cererea primită în cererea procesată.
2. **StartMisca**: Metoda startMoving din StartMovingService inițiază mișcarea liftului.
3. **Ajuns**: Metoda handleArrival din ArrivalService marchează sosirea liftului la un etaj.
4. **ActualizareEtaj**: Metoda updateCurrentFloor din FloorService actualizează etajul curent.

**Arce**:

1. **CererePrimita** -> **PrimireCerere**
2. **PrimireCerere** -> **ProcesareCerere**
3. **ProcesareCerere** -> **StartMisca**
4. **StartMisca** -> **LiftInMiscare**
5. **LiftInMiscare** -> **Ajuns**
6. **Ajuns** -> **AjunsLaDestinatie**
7. **AjunsLaDestinatie** -> **ActualizareEtaj**
8. **ActualizareEtaj** -> **EtajCurent**

**3. Diagrama de Clase pentru O Componentă Selectată**

Vom alege **clasa Request** pentru a exemplifica diagrama de clase.

**Diagrama de Clase pentru Request**:

* **Request**: Reprezintă cererile utilizatorilor pentru mișcarea liftului.

+----------------------+

| Request |

+----------------------+

| - id: Long |

| - req\_up: boolean |

| - req\_down: boolean |

+----------------------+

| +getId(): Long |

| +setId(id: Long): void|

| +isReqUp(): boolean |

| +setReqUp(req\_up: boolean): void|

| +isReqDown(): boolean|

| +setReqDown(req\_down: boolean): void|

+----------------------+

**Explicație**:

* **Atribute**:
  + id: Identificator unic pentru fiecare cerere.
  + req\_up: Boolean care indică dacă cererea este pentru mișcare în sus.
  + req\_down: Boolean care indică dacă cererea este pentru mișcare în jos.
* **Metode**:
  + Getters și setters pentru fiecare atribut pentru a accesa și modifica datele.

**4. Secvențele de Cod Relevante**

Pentru a implementa componenta Request, vom prezenta secvențele de cod relevante pentru crearea și manipularea cererilor.

**1. Definirea Entității Request**:

package com.lift\_control.liftcontrol.entity;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

@Entity

public class Request {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private Long liftId;

private boolean reqUp;

private boolean reqDown;

// Getters and Setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public Long getLiftId() {

return liftId;

}

public void setLiftId(Long liftId) {

this.liftId = liftId;

}

public boolean isReqUp() {

return reqUp;

}

public void setReqUp(boolean reqUp) {

this.reqUp = reqUp;

}

public boolean isReqDown() {

return reqDown;

}

public void setReqDown(boolean reqDown) {

this.reqDown = reqDown;

}

}

**2. Crearea Repository-ului RequestRepository**:

package com.lift\_control.liftcontrol.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.lift\_control.liftcontrol.entity.Request;

public interface RequestRepository extends JpaRepository<Request, Long> {

}**3. Crearea Serviciului RequestService**:

package com.lift\_control.liftcontrol.service;

import com.lift\_control.liftcontrol.entity.Request;

import com.lift\_control.liftcontrol.repository.RequestRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class RequestService implements RequestServiceInterface {

@Autowired

private RequestRepository requestRepository;

@Override

public List<Request> getAllRequests() {

return requestRepository.findAll();

}

@Override

public Request getRequestById(Long id) {

return requestRepository.findById(id).orElse(null);

}

@Override

public Request saveRequest(Request request) {

return requestRepository.save(request);

}

@Override

public void deleteRequest(Long id) {

requestRepository.deleteById(id);

}

}**4. Crearea Controller-ului RequestController**:

package com.lift\_control.liftcontrol.Controller;

import com.lift\_control.liftcontrol.dto.RequestDto;

import com.lift\_control.liftcontrol.entity.Request;

import com.lift\_control.liftcontrol.service.RequestService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/api/request")

public class RequestController {

@Autowired

private RequestService requestService;

@GetMapping

public List<Request> getAllRequests() {

return requestService.getAllRequests();

}

@PostMapping

public Request createRequest(@RequestBody RequestDto requestDto) {

Request request = new Request();

request.setReqUp(requestDto.isReqUp());

request.setReqDown(requestDto.isReqDown());

return requestService.saveRequest(request);

}

@DeleteMapping("/{id}")

public void deleteRequest(@PathVariable Long id) {

requestService.deleteRequest(id);

}

}

**5. Analiza Cerințelor Temporale**

**Analiza temporară** se referă la evaluarea și asigurarea că sistemul poate respecta toate cerințele de timp specificate, cum ar fi timpul maxim de răspuns pentru cereri și perioada în care liftul trebuie să ajungă la un etaj.

* **Cerințe Temporale**: Determină cât de repede trebuie să răspundă sistemul la cereri (de exemplu, liftul trebuie să ajungă la un etaj în 30 de secunde).
* **Măsurare Performanță**: Se pot utiliza unelte de monitorizare pentru a măsura timpul de răspuns al sistemului și a asigura respectarea cerințelor temporale.

**Teorie: Explicații și Exemplificări**

**1. Polling vs. Interrupts**

* **Polling**: Verificarea periodică a unei condiții sau a unui eveniment. De exemplu, un sistem care verifică în mod regulat dacă există cereri de mișcare a liftului. Este simplu de implementat dar poate fi ineficient, deoarece necesită verificări constante care consumă resurse.
* **Interrupts**: Sistemul este notificat automat atunci când apare un eveniment. De exemplu, liftul poate fi notificat automat când apare o cerere de mișcare. Este mai eficient și mai rapid decât polling-ul, deoarece sistemul nu trebuie să verifice constant condiția, ci doar răspunde atunci când apare evenimentul.

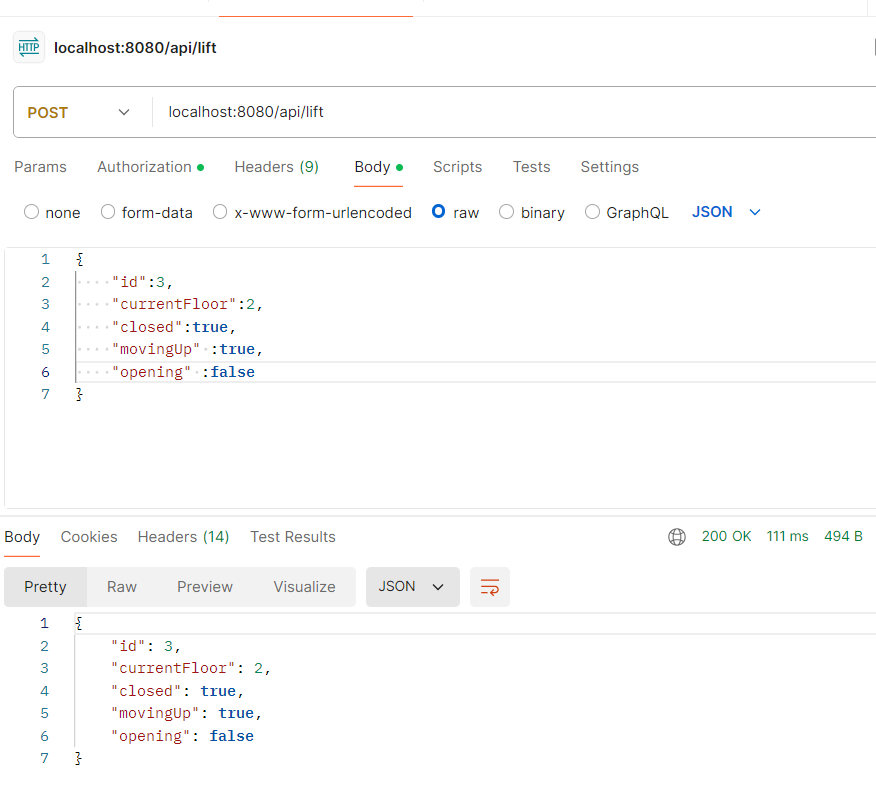
**2. Relația Dintre Viteza Liftului și Deadline**

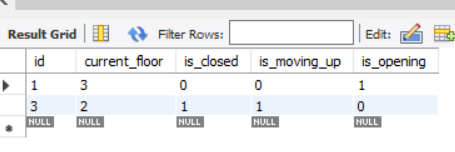
* **Viteza Liftului**: Viteza cu care liftul se mișcă afectează direct capacitatea de a respecta termenele limită. Dacă liftul se mișcă prea lent, nu va putea ajunge la etaje în timpul specificat.
* **Deadline**: Este timpul maxim permis pentru completarea unei sarcini (de exemplu, timpul necesar pentru ca liftul să ajungă la un etaj). Viteza liftului trebuie să fie ajustată astfel încât să se încadreze în aceste limite de timp.

**3. Modul de Realizare a Analizei Temporare**

* **Definirea Cerințelor Temporale**: Identifică toate cerințele legate de timp (ex. liftul trebuie să ajungă la etajul solicitat în mai puțin de 30 de secunde).
* **Măsurarea Performanței**: Utilizează unelte pentru a măsura efectiv timpul de răspuns și a verifica dacă cerințele sunt îndeplinite.
* **Optimizare**: Dacă timpul de răspuns nu respectă cerințele, ajustează parametrii sistemului sau optimizează componentele pentru a îmbunătăți performanța.

**Testare și Depanare**





**Testare**

1. **Testare Unități**:
   * **Ce am făcut**: Am verificat fiecare parte a aplicației, cum ar fi serviciile și repository-urile, pentru a mă asigura că funcționează așa cum ar trebui. Am scris teste pentru fiecare metodă importantă pentru a verifica dacă logica funcționează corect.
   * **Cum am procedat**: Am folosit instrumente de testare unități care au permis simularea comportamentului diferitelor componente, fără a interacționa cu restul aplicației. De exemplu, am testat dacă serviciul care gestionează cererile funcționează corect atunci când primește date corecte sau incorecte.
2. **Testare Integrare**:
   * **Ce am făcut**: Am verificat dacă toate părțile aplicației lucrează bine împreună. De exemplu, am testat endpoint-urile API pentru a vedea dacă răspund corect la cererile făcute de utilizatori.
   * **Cum am procedat**: Am folosit Postman pentru a trimite cereri către serverul aplicației și am verificat răspunsurile. Am verificat dacă datele se transmit corect între frontend și backend și dacă aplicația se comportă conform așteptărilor.
3. **Testare Manuală**:
   * **Ce am făcut**: Am testat aplicația așa cum ar face-o un utilizator obișnuit pentru a mă asigura că toate funcționalitățile sunt ușor de folosit și funcționează bine.
   * **Cum am procedat**: Am interacționat manual cu aplicația, trimițând cereri de mișcare pentru lift și verificând dacă liftul se mișcă așa cum ar trebui și dacă utilizatorul poate vedea starea liftului în timp real.

**Depanare**

* **Ce am făcut**: Când am întâmpinat probleme, am identificat sursa erorilor și am corectat-o.
* **Cum am procedat**: Am folosit instrumente de depanare pentru a urmări codul și a înțelege unde apar problemele. Am analizat mesajele de eroare, am verificat fluxul de execuție al aplicației și am făcut ajustări necesare pentru a remedia problemele.
* .

### Dezvoltarea Interfeței Front-End a Aplicației de Control al Liftului

#### Introducere

Pentru a completa funcționalitatea aplicației de control al liftului, am dezvoltat o interfață front-end care permite utilizatorilor să interacționeze cu sistemul de lifturi în timp real. Această interfață a fost creată utilizând HTML, CSS și JavaScript, și a fost concepută pentru a oferi o experiență interactivă și intuitivă. În continuare, voi detalia tehnologiile utilizate, implementarea funcționalităților principale și metodele de testare folosite pentru a asigura corectitudinea și eficiența sistemului.

#### Tehnologii Folosite

* **HTML și CSS**: Am folosit HTML pentru a structura paginile și CSS pentru a stiliza interfața utilizatorului. Am utilizat Flexbox și Grid Layout pentru a organiza componentele vizuale și pentru a asigura o afișare coerentă pe diferite dimensiuni de ecran.
* **JavaScript**: JavaScript a fost utilizat pentru a gestiona interacțiunile dinamice și pentru a comunica cu partea de backend a aplicației. Prin intermediul AJAX și Fetch API, am putut trimite cereri către server și am primit răspunsuri în timp real.
* **Font Awesome**: Am integrat biblioteca Font Awesome pentru a adăuga pictograme (de exemplu, săgeți pentru butoanele de control al liftului) care să îmbunătățească interfața utilizatorului.

#### Funcționalități Implementate

1. **Controlul Liftului**:
   * Fiecare lift din interfață este asociat cu un set de butoane care permit utilizatorilor să solicite liftul la un anumit etaj.
   * Când utilizatorul apasă pe un buton, o cerere este trimisă către backend folosind JavaScript, care inițiază mișcarea liftului.
2. **Simularea Mișcării Liftului**:
   * Interfața simulează vizual mișcarea liftului prin animarea poziției acestuia pe axa verticală. Această animație este declanșată după primirea răspunsului de la backend, care confirmă că liftul a început să se deplaseze.
3. **Monitorizarea Sosirilor**:
   * După ce liftul ajunge la etajul solicitat, interfața actualizează automat statusul pentru a reflecta sosirea liftului. Aceasta se face prin actualizarea vizuală a poziției liftului și prin afișarea unui mesaj de confirmare către utilizator.

#### Integrarea cu Backend-ul

Fiecare interacțiune a utilizatorului cu interfața front-end este conectată direct cu backend-ul aplicației:

* **Trimite Cererea de Mișcare**: Când un utilizator apasă un buton pentru a solicita liftul la un anumit etaj, JavaScript trimite o cerere POST către un endpoint specific din backend (de exemplu, /api/lift/move), incluzând informații despre etajul solicitat și direcția de deplasare.
* **Înregistrarea în Baza de Date**: Backend-ul primește cererea și o înregistrează în baza de date MySQL, actualizând starea liftului, incluzând timestamp-ul pentru momentul în care mișcarea a fost inițiată (startMoving), etajul țintă (requestFloor), și momentul sosirii liftului (arrival).
* **Actualizarea Interfeței**: După procesarea cererii în backend și actualizarea bazei de date, interfața front-end primește un răspuns care declanșează animarea mișcării liftului și actualizarea mesajelor afișate.

