Name: Nirav Parekh , 929750

Questions:-

1. Design backend REST APIs using Spring Boot Application eg: Employee CRUD
2. Design frontend for existing REST APIs using React
3. Add validations to design forms
4. Add exception handling to your Spring Boot REST APIs
5. Add Swagger Documentation to your Spring Boot App
6. Add Spring Security to your project.
7. Add role based Access Control to APIs as below - 1. Role [USER] can have to View Access only 2. Role [DEV] can Create and Update operation 3. Role [ADMIN] can perform delete operations only.

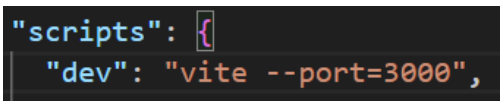
How to run the project:

To Run the backend:

1. Use Intellij
2. Change the password of MySQL DB. Also change db if needed
3. Go inside src -> main -> java -> com.app -> Application.java. Then right click and run Application.main

To run frontend:

1. npm install (to install node modules.)
2. Inside package.json change:



3. Then npm run dev

1. Employee REST APIs using Spring Boot application

Controller

API:

1. Get /all : To get list of employees
2. Get /id : To get an employee by ID
3. Post /add : To add an employee
4. Put /update : To update an employee
5. Delete /delete: To Delete an employee

Explanation:

1. Get All Employees:
    - URL: GET /employees
    - Returns a list of employees.
2. Get Employee by ID:
    - URL: GET /employees/{id}
    - Returns a specific employee based on the provided ID.
3. Add New Employee:
    - URL: POST /employees
    - Adds a new employee and returns the created employee.
4. Update Employee:
    - URL: PUT /employees/{id}
    - Updates an existing employee based on the provided ID and request body.
5. Delete Employee:
    - URL: DELETE /employees/{id}
    - Deletes a specific employee based on the provided ID.

Model

Model has the following Fields:-

1. id
2. firstName
3. lastName
4. city
5. salary
6. email
7. password
8. dept

1. id:
    - Type: Long
    - Description: Unique identifier for the employee, auto-generated.
    - Annotations: @Id, @GeneratedValue(strategy = GenerationType.IDENTITY), @Column(name = "ID")
2. firstName:
    - Type: String
    - Description: Employee's first name.
    - Constraints: Cannot be blank, maximum length of 20 characters.

- Annotations: @NotBlank(message = "First Name Can't be Empty"), @Column(name = "FirstName", nullable = false, length = 20)

3. lastName:
    - Type: String
    - Description: Employee's last name.
    - Constraints: Cannot be blank, maximum length of 20 characters.
    - Annotations: @NotBlank(message = "Last Name Can't be Empty"), @Column(name = "LastName", nullable = false, length = 20)
4. city:
    - Type: String
    - Description: City where the employee resides.
    - Constraints: Cannot be blank, maximum length of 20 characters.
    - Annotations: @NotBlank(message = "City Can't be Empty"), @Column(name = "City", nullable = false, length = 20)
5. salary:
    - Type: Double
    - Description: Employee's salary.
    - Constraints: Cannot be blank.
    - Annotations: @NotBlank(message = "Salary Can't Be Empty"), @Column(name = "Salary", nullable = false, length = 20)
6. email:
    - Type: String
    - Description: Employee's email address.
    - Constraints: Must be a valid email format, cannot be blank, must be unique.
    - Annotations: @Email(message = "Email should be valid"), @NotBlank(message = "Email can't be Empty"), @Column(name = "Email", unique = true, nullable = false)
7. password:
    - Type: String
    - Description: Employee's password.
    - Constraints: Cannot be blank.
    - Annotations: @NotBlank(message = "Password is Required")
8. dept:
    - Type: String
    - Description: Department where the employee works.
    - Constraints: Cannot be blank, maximum length of 20 characters.

- Annotations: @NotBlank(message = "Enter Department"), @Column(name = "Dept", nullable = false, length = 20)

Lombok Annotations:

- @NoArgsConstructor: Generates a no-argument constructor.
- @AllArgsConstructor: Generates an all-argument constructor.
- @Getter: Generates getters for all fields.
- @Setter: Generates setters for all fields.
- @ToString(exclude = "password"): Generates a toString method excluding the password field.

JPA Annotations:

- @Entity: Specifies that the class is an entity.
- @Table(name = "Emp_Tbl"): Specifies the table name in the database.
- @Id: Specifies the primary key.
- @GeneratedValue(strategy = GenerationType.IDENTITY): Specifies the primary key generation strategy.
- @Column: Customizes the mapping between the field and the database column.

Validation Annotations:

- @NotBlank: Ensures the field is not null and the trimmed length is greater than zero.
- @Email: Ensures the field contains a valid email address.

2. Design frontend for existing REST APIs using React

So I have designed a front-end for my application using React.
I have used axios to fetch the data:

```
useEffect(() => {
  const fetchData = async () => {
    setIsLoading(true);
    setError(null);

    try {
      const response = await axios.get('http://localhost:8080/employee');
      setEmployees(response.data);
    } catch (error) {
      console.error('Error fetching data:', error);
      setError(error);
    } finally {
      setIsLoading(false);
    }
  };

  fetchData();
}, [history]);
```

Tailwind for styling. And React-router for Routing:

```
<Switch>
  <Route exact path="/" component={List} />
  <Route path="/add" component={Add} />
  <Route path="/update/:id" component={Update} />
  <Route path="/delete" component={Delete} />
</Switch>
```

3. Add validations to design forms

For validation I have basically added some if else statement that will be checked before form is submitted

```
const validate = () => {
  const newErrors = {};
  if (!formData.firstName) newErrors.firstName = 'First name is required';
  if (!formData.lastName) newErrors.lastName = 'Last name is required';
  if (!formData.city) newErrors.city = 'City is required';
  if (!formData.salary || formData.salary <= 0) newErrors.salary = 'Salary must be a positive number';
  if (!formData.email) newErrors.email = 'Email is required';
  else if (!/\S+@\S+\.\S+/.test(formData.email)) newErrors.email = 'Email is invalid';
  if (!formData.password) newErrors.password = 'Password is required';
  else if (formData.password.length < 6) newErrors.password = 'Password must be at least 6 characters';
  if (!formData.dept) newErrors.dept = 'Department is required';
  return newErrors;
};
```

When we try to add employee using POST request it'll check for errors:-

```
const handleSubmit = async (e) => {
  e.preventDefault();
  const newErrors = validate();
  if (Object.keys(newErrors).length > 0) {
    setErrors(newErrors);
    return;
  }
  try {
    const response = await axios.post(
      'http://localhost:8080/employee',
      formData
    );
    setShowNotification(true);
    setErrorMessage('');
    setTimeout(() => {setShowNotification(false);}, 3000);
  } catch (error) {
    setErrorMessage(error.response?.data?.message || 'An error occurred');
  }
};
```

List Employees Add Employee Delete Employee

ID	First Name	Last Name	City	Salary	Email	Department	Actions
2	Jane	Smith	Los Angeles	60000	jane.smith@example.com	HR	Update
3	Michael	Johnson	Chicago	55000	michael.johnson@example.com	Finance	Update
4	Nirav	Parekh	Pune	50000	niruparekh09@gmail.com	IT	Update
5	Jigar	Singh	Raipur	0	jigar@gmail.com	IT	Update
18	James	Michaels	LA	47500	James.Michaels@outlook.com	Finance	Update
19	Jack	Michaels	LA	78500	Jack.Michaels@outlook.com	Finance	Update

List Employees Add Employee Delete Employee

First Name:

Last Name:

City:

Salary:

0

Email:

Password:

Department:

Add Employee

List Employees Add Employee Delete Employee

ID	FIRST NAME	LAST NAME	CITY	SALARY	EMAIL	DEPARTMENT	DELETE
2	Jane	Smith	Los Angeles	60000	jane.smith@example.com	HR	Delete
3	Michael	Johnson	Chicago	55000	michael.johnson@example.com	Finance	Delete
4	Nirav	Parekh	Pune	50000	niruparekh09@gmail.com	IT	Delete
5	Jigar	Singh	Raipur	0	jigar@gmail.com	IT	Delete
18	James	Michaels	LA	47500	James.Michaels@outlook.com	Finance	Delete
19	Jack	Michaels	LA	78500	Jack.Michaels@outlook.com	Finance	Delete

List Employees Add Employee Delete Employee

First Name:

Jane

Last Name:

Smith

City:

Los Angeles

Salary:

60000

Email:

jane.smith@example.com

Password:

Department:

HR

Update Employee

4. Exception Handling in REST APIs

APIResponse DTO for custom messages:-

```
@NoArgsConstructor  3 usages  new *
@Getter
@Setter
@ToString
public class APIResponse {
    private LocalDateTime timeStamp;
    private String message;

    public APIResponse(String message) {  2 usages  new *
        super();
        this.message = message;
        this.timeStamp = LocalDateTime.now();
    }
}
```

ResourceNotFoundException for throwing custom exceptions:-

```
public class ResourceNotFoundException extends RuntimeException {  7 usages  Nirav Parekh
    public ResourceNotFoundException(String message) {  3 usages  Nirav Parekh
        super(message);
    }
}
```

GlobalExceptionHandler to handle all exception that can occur:-

```
@RestControllerAdvice  no usages  new *
public class GlobalExceptionHandler {
    @ExceptionHandler(MethodArgumentNotValidException.class)  no usages  new *
    public ResponseEntity<?> handleMethodArgumentNotValidException
    (MethodArgumentNotValidException e) {
        System.out.println("in meth arg invalid " + e);
        List<FieldError> errList = e.getFieldErrors();
        Map<String, String> map = errList.stream().collect(Collectors.toMap(FieldError::getField, FieldError::getDefaultMessage));
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(map);
    }

    @ExceptionHandler(ResourceNotFoundException.class)  no usages  new *
    public ResponseEntity<?> handleMotoGPException
    (ResourceNotFoundException e) {
        System.out.println("in res not found exc");
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(new APIResponse(e.getMessage()));
    }

    @ExceptionHandler(Exception.class)  no usages  new *
    public ResponseEntity<?> handleException
    (Exception e) {
        System.out.println("in catch-all  exc");
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(new APIResponse(e.getMessage()));
    }
}
```

Annotations used:-

1. @RestControllerAdvice:
    - This annotation is used to define a global exception handler for all controllers in the Spring application.

- It allows you to handle exceptions across the whole application in a centralised manner.

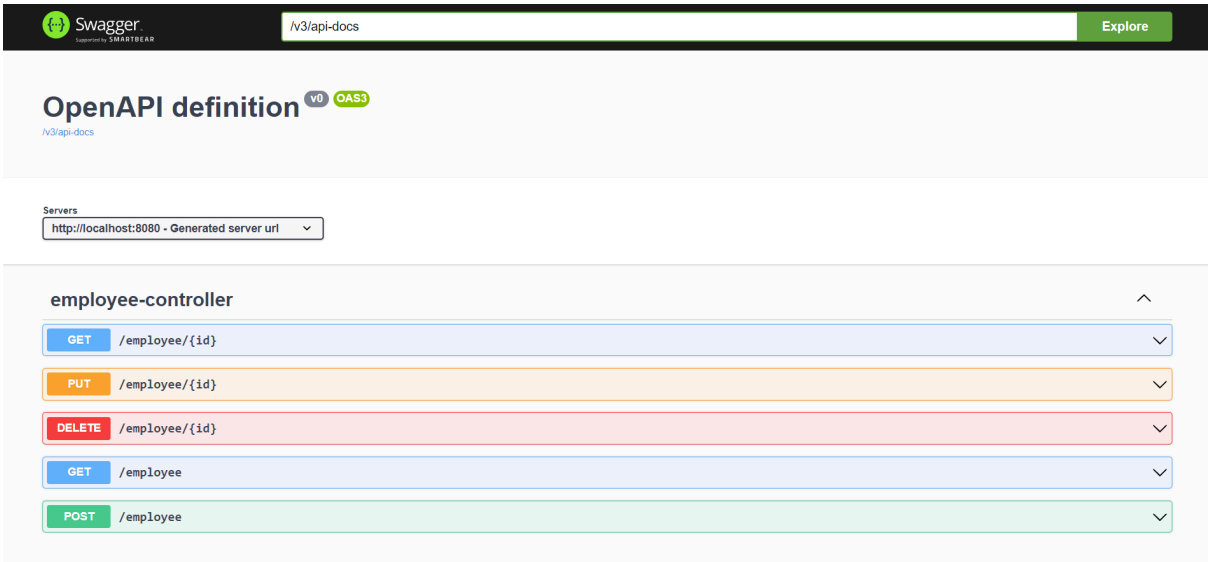
2. @ExceptionHandler(MethodArgumentNotValidException.class):
   - This annotation is used to define a method that handles exceptions of a specific type, in this case, MethodArgumentNotValidException.
   - When this exception occurs, Spring will invoke the method handleMethodArgumentNotValidException() to handle it.
3. @ExceptionHandler(ResourceNotFoundException.class):
   - Similar to the previous annotation, this defines a method to handle exceptions of type ResourceNotFoundException.
4. @ExceptionHandler(Exception.class):
   - This annotation defines a method to handle any other exception that is not explicitly handled by other @ExceptionHandler methods.

5. Adding swagger documentation in our REST API

To add swagger documentation we can simply add a maven dependency provided to us by openAPI:

```
<!-- https://mvnrepository.com/artifact/org.springdoc/springdoc-openapi-ui -->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
    <version>1.7.0</version>
</dependency>
```

And then once we run our application, swagger ui can be accessed in:
http://localhost:8080/swagger-ui/index.html
And it looks like this:



Here we can Test out APIs.

6. Add Spring Security to your project

Just add two Maven dependencies to enable spring security for your project:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

spring-boot-starter-security:

- This starter aggregates Spring Security-related dependencies.
- It provides the necessary components to integrate Spring Security into your Spring Boot application.
- When you include this dependency, Spring Boot sets up basic security configurations for your application.
- It includes features like authentication, authorization, and secure endpoints.
- You can customise the security settings further by adding your own configuration classes.

Now we can also add BCryptPasswordEncoder, which will help us encoding our password so that we don't need to store the password of the user in the database normally. Using this we can securely hash and encode the password.

```
@Bean  no usages
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

```
@Override  1 usage
public EmployeeResponse addEmployee(EmployeeInsert emp) {
    Employee employee = modelMapper.map(emp, Employee.class);
    // Manually set the Role
    employee.setRole(Role.valueOf(emp.getRole().toUpperCase()));
    employee.setPassword(encoder.encode(emp.getPassword()));
    employee = employeeRepository.save(employee);
    return modelMapper.map(employee, EmployeeResponse.class);
}
```

8. Add role based Access Control to APIs as below - 1. Role [USER] can have to View Access only 2. Role [DEV] can Create and Update operation 3. Role [ADMIN] can perform delete operations only.

For this you need to add Security Filter Chain:-

```
@Bean  no usages
public SecurityFilterChain configureAuthorization(HttpSecurity httpSecurity) throws Exception {
    httpSecurity.authorizeRequests() ExpressionInterceptUrlRegistry
            .antMatchers( ...antPatterns: "/employee/list/**","/employee/list").hasRole("USER")
            .antMatchers( ...antPatterns: "/employee/add","/employee/update").hasRole("DEV")
            .antMatchers( ...antPatterns: "/employee/delete").hasRole("ADMIN")
            .anyRequest().authenticated()
            .and() HttpSecurity
            .httpBasic();
    return httpSecurity.build();
}
```

Via this you can add authorization rules for different end points.
As given in the task:

1. Endpoints for list of employees and information of specific employees is accessible by a USER only.
2. Endpoints for Adding and Updating an Employee is given to a DEV only.
3. Endpoint for deleting an employee is given to ADMIN only.

To check whether the user is a USER, DEV or ADMIN, We'll be using service based authorization. For this we need to create class that implements UserDetailsService which will fetch the user by a unique identifier (email in our case) and then send to the class that implements UserDetails to verify it's role and password (which is hashed using BCrypt).

```
public class CustomEmployeeDetailService implements UserDetailsService {  no usages

    @Autowired  1 usage
    EmployeeRepository employeeRepository;

    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
        Employee authEmp = employeeRepository.findByEmail(email)
                .orElseThrow(() -> new UsernameNotFoundException("User with Email not found"));
        return new CustomEmployeeDetails(authEmp);
    }
}
```

```java
public class CustomEmployeeDetails implements UserDetails {  1 usage
    private Employee authEmpDetails;  4 usages

    public CustomEmployeeDetails(Employee authEmp) {  1 usage
        super();
        this.authEmpDetails = authEmp;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority(authEmpDetails.getRole().name()));
    }

    @Override
    public String getPassword() {
        return authEmpDetails.getPassword();
    }

    @Override
    public String getUsername() {
        return authEmpDetails.getEmail();
    }
```