1. Introduction

The backend of the Logistics Management System is a critical component responsible for handling data management, user authentication, logistics operations, and secure communication between the client and server. Built using Spring Boot, a popular Java framework for building microservices and RESTful APIs, this report provides an in-depth analysis of the backend architecture, including configuration, controller services, and security mechanisms.

2. Architecture and Components

The backend architecture is composed of several key components, including:

- **Spring Boot Framework**: Utilized for building a robust RESTful API with support for dependency injection, security, and data management.
- **MySQL Database:** Used as the primary relational database for storing user data, logistics records, inventory details, contact information, and other application-related information.
- **Spring Security:** Provides authentication and authorization functionalities, ensuring secure access to the application's resources.
- **JWT (JSON Web Token):** Used for user authentication, providing a secure and scalable solution for managing user sessions.
- **Service Layer:** Handles business logic and interaction between the controllers and repositories.
- **Repository Layer:** Responsible for data access and persistence, leveraging JPA (Java Persistence API) for database interactions.

3. Components Description

3.1. Configuration

The configuration files define the overall behavior of the backend, including CORS (Cross-Origin Resource Sharing) settings, security configurations, and database connections.

- **CorsConfig:** Manages CORS settings, allowing the frontend application hosted on http://localhost:3000 to interact with the backend services.
- **SecurityConfig:** Configures the security settings, permitting public access to specific endpoints such as user registration, login, and logistics operations APIs, while restricting access to other resources.

3.2. Controllers

Controllers are responsible for handling HTTP requests, processing data, and returning appropriate responses.

- **ShipmentController:** Manages logistics operations, allowing users to create, view, and update logistics records.
- **UserController:** Handles user-related operations, including registration, login, and user management.
- **InventoryController:** Manages inventory operations, allowing users to add, view, and update inventory records.
- **ContactController:** Handles contact information, allowing users to manage contact records related to logistics operations.

3.3. Services

The service layer contains business logic and interacts with the repository layer to perform CRUD (Create, Read, Update, Delete) operations.

- **ShipmentService:** Provides methods to add, retrieve, and update logistics records.

- **UserService:** Manages user registration, authentication, and updates, utilizing password encryption and JWT generation.
- **InventoryService:** Manages inventory records, providing methods to add, view, and update inventory details.
- **ContactService**: Handles contact records, providing methods to manage contact information.

3.4. Models

The models represent the database entities and are annotated with JPA annotations for ORM (Object-Relational Mapping).

- **Shipment:** Represents a logistics record entity, with attributes like shipment ID, origin, destination, status, estimated delivery time, and carrier information.
- **User:** Represents a user entity, containing attributes such as username, password, email, phone, and address.
- **Inventory:** Represents an inventory entity, with attributes like inventory ID, item name, quantity, location, and status.
- **Contact:** Represents a contact entity, containing attributes such as contact ID, name, phone, email, and related shipment or user information.

3.5. Repositories

Repositories interface with the database, allowing the application to perform CRUD operations seamlessly.

- **ShipmentRepository:** Extends JpaRepository to provide data access for logistics records.
- **UserRepository:** Extends JpaRepository to provide data access for user information.

- **InventoryRepository:** Extends JpaRepository to provide data access for inventory records.
- **ContactRepository:** Extends JpaRepository to provide data access for contact information.

3.6. Security

Security is implemented using Spring Security and JWT for authentication.

- **BCryptPasswordEncoder**: Encrypts user passwords before storing them in the database.
- **JwtUtil**: Handles JWT generation, extraction of claims, and token validation.

Fig 3.6.1 Backend System Architecture

4. Entity-Relationship

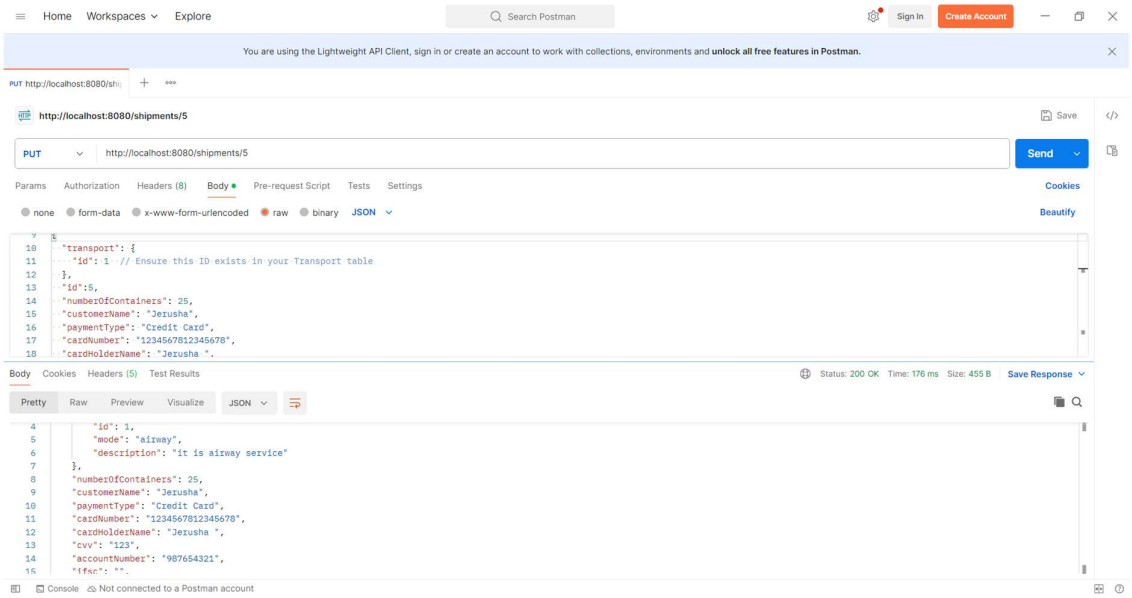
4.1.Models

The models represent the database entities and are annotated with JPA annotations for ORM (Object-Relational Mapping).

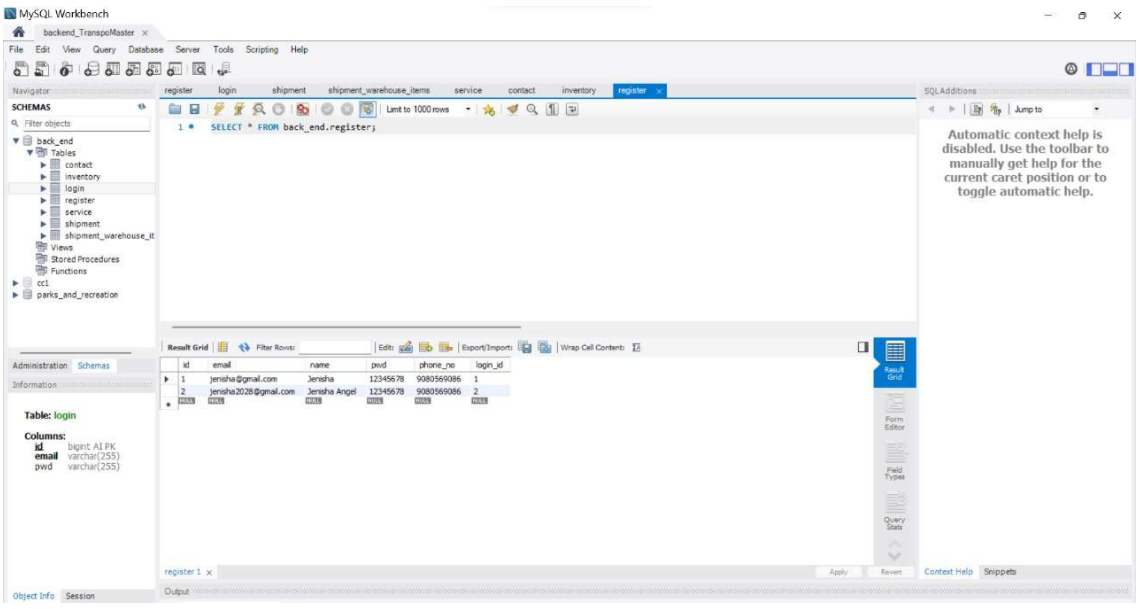
- **User:** Represents a user entity, containing attributes such as:
    - **id:** Long (Primary Key)
    - **username:** String
    - **password:** String
    - **email:** String
    - **phone:** String
    - **address:** String
- **Shipment:** Represents a logistics record entity, with attributes like:
    - **id:** Long (Primary Key)
    - **shipmentId:** String
    - **origin:** String
    - **destination:** String
    - **status:** String
    - **estimatedDeliveryTime:** String
    - **carrierInformation:** String
- **Inventory:** Represents an inventory entity, with attributes like:
    - **id:** Long (Primary Key)
    - **inventoryId:** String
    - **itemName:** String

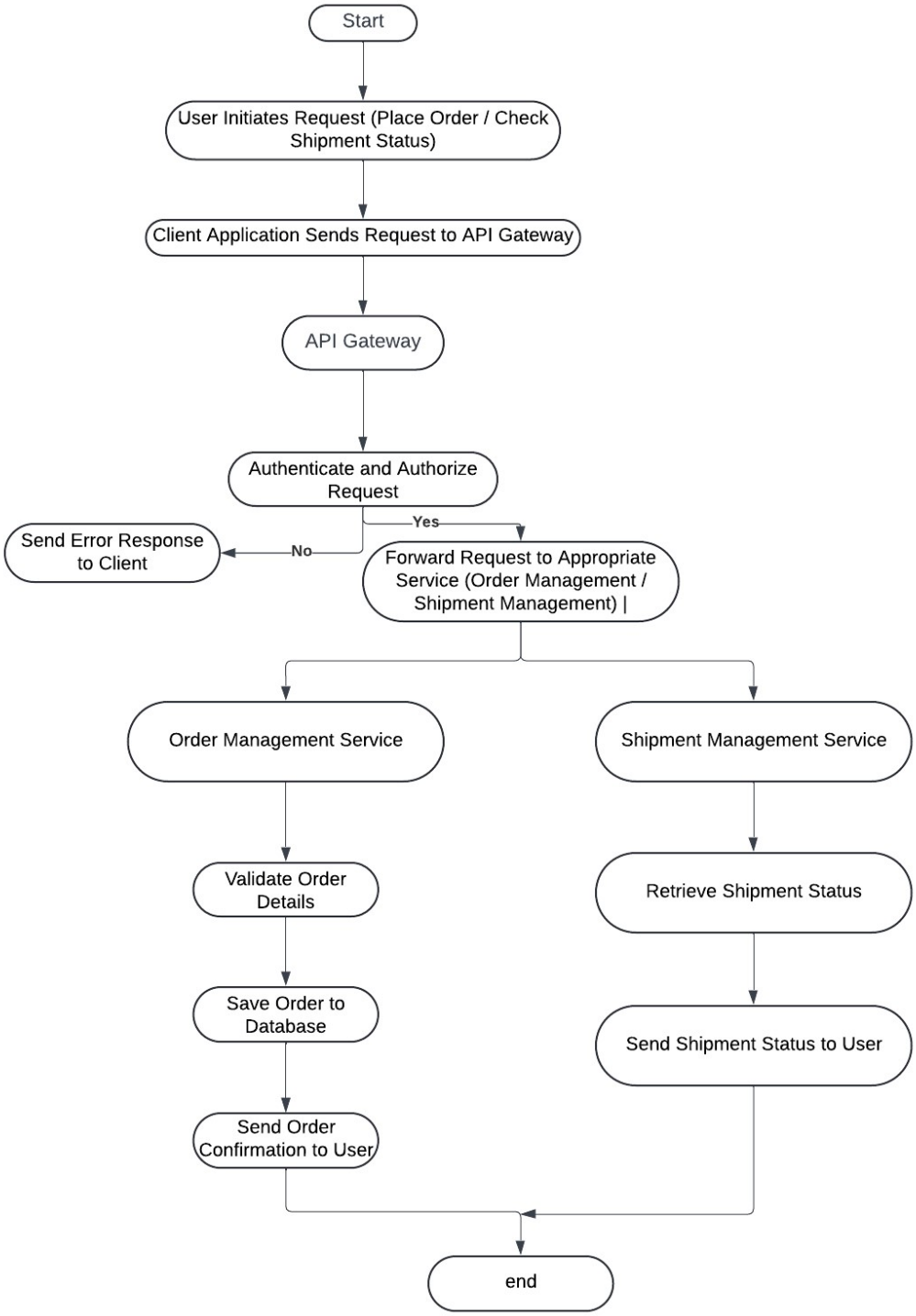
    - **quantity:** Integer
    - **location**: String
    - **status:** String
- **Contact:** Represents a contact entity, containing attributes such as:
    - **id:** Long (Primary Key)
    - **contactId:** String
    - **name:** String
    - **phone:** String
    - **email:** String
    - **relatedShipmentId:** Long (Foreign Key referencing Shipment)
    - **relatedUserId:** Long (Foreign Key referencing User)

4.4. Postman Api



4.6. Mysql Database





4.7. REST API FlOWCHART

5. Coding

5.1. UserController.java

```
package com.example.backend.controller;

import com.example.backend.model.Register;

import com.example.backend.service.UserService;

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

import org.springframework.http.ResponseEntity;

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

import java.util.List;

import java.util.Optional;

@RestController

@RequestMapping("/api/users")

public class UserController {

    @Autowired

    private UserService userService;

    @PostMapping

    public ResponseEntity<Register> createRegister(@RequestBody Register register)
{

        try {

            Register newRegister = userService.createRegister(register);

            return ResponseEntity.ok(newRegister);

        } catch (Exception e) {

            return ResponseEntity.badRequest().body(null);
```

```
        }
    }
    @GetMapping
    public ResponseEntity<List<Register>> getAllRegisters() {
        List<Register> registers = userService.getAllRegisters();
        return ResponseEntity.ok(registers);
    }
    @GetMapping("/{id}")
    public ResponseEntity<Register> getRegisterById(@PathVariable Long id) {
        Optional<Register> register = userService.getRegisterById(id);
        return register.map(ResponseEntity::ok).orElseGet(() ->
ResponseEntity.notFound().build());
    }
    @PutMapping("/{id}")
    public ResponseEntity<Register> updateRegister(@PathVariable Long id,
@RequestBody Register updatedRegister) {
    Optional<Register> updated = userService.updateRegister(id, updatedRegister);
    return updated.map(ResponseEntity::ok).orElseGet(() ->
ResponseEntity.notFound().build());
    }
    @DeleteMapping
    public ResponseEntity<Void> deleteAllRegisters() {
        userService.deleteAllRegisters();
        return ResponseEntity.noContent().build();
```

```
    }
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteLoginById(@PathVariable Long id) {
        try {
            userService.deleteLoginById(id);
            return ResponseEntity.noContent().build();
        } catch (Exception e) {
            return ResponseEntity.notFound().build();
        }
    }
}
```

5.2. Login.java

```
package com.example.backend.model;
import jakarta.persistence.*;
@Entity
public class Login {
    @Id
    private Long id;
    @Column(name = "email", unique = true)
    private String email;
    @Column(name = "pwd")
    private String password;
```

```
// Getters and Setters
public Long getId() {
    return id;
}
pubic void setId(Long id) {
    this.id = id;
}
public String getEmail() {
    return email;
}
public void setEmail(String email) {
    this.email = email;
}
public String getPassword() {
    return password;
}
public void setPassword(String password) {
    this.password = password;
}
// Default constructor
public Login() {}
// Parameterized constructor
public Login(Long id, String email, String password) {
```

```
        this.id = id;
        this.email = email;
        this.password = password;
    }
}
```

5.3.Register.java

```
package com.example.backend.model;
import jakarta.persistence.*;
@Entity
public class Register {
    @Id
    // @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "name")
    private String name;
    @Column(name = "phone_no")
    private String phone;
    @Column(name = "email", unique = true)
    private String email;
    @Column(name = "pwd")
    private String password;
    @OneToOne(cascade = CascadeType.ALL)
```

```
@JoinColumn(name = "login_id")
private Login login;
// Getters and Setters
public Long getId() {
    return id;
}
public void setId(Long id) {
    this.id = id;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public String getPhone() {
    return phone;
}
public void setPhone(String phone) {
    this.phone = phone;
}
public String getEmail() {
    return email;
```

```
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public Login getLogin() {
        return login;
    }
    public void setLogin(Login login) {
        this.login = login;
    }
    // Default constructor
    public Register() {}
    // Parameterized constructor
    public Register(Long id, String name, String phone, String email, String password,
Login login) {
        this.id = id;
        this.name = name;
```

```
        this.phone = phone;
        this.email = email;
        this.password = password;
        this.login = login;
    }
}
```

5.4. LoginRepository.java

```
package com.example.backend.repository;
import com.example.backend.model.Login;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import java.util.Optional;
@Repository
public interface LoginRepository extends JpaRepository<Login, Long> {
    Optional<Login> findByEmail(String email);
}
```

5.5. RegisterRepository.java

```
package com.example.backend.repository;
import com.example.backend.model.Register;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
```

```
@Repository
public interface RegisterRepository extends CrudRepository<Register, Long> {
}
```

5.6. UserService.java

```
package com.example.backend.service;
import com.example.backend.model.Login;
import com.example.backend.model.Register;
import com.example.backend.repository.LoginRepository;
import com.example.backend.repository.RegisterRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.StreamSupport;
@Service
public class UserService {
    @Autowired
    private LoginRepository loginRepository;
    @Autowired
    private RegisterRepository registerRepository;
```

```
    @Transactional
    public Register createRegister(Register register) {
        Optional<Login> existingLogin =
loginRepository.findByEmail(register.getLogin().getEmail());
        if (existingLogin.isPresent()) {
            register.setLogin(existingLogin.get());
        } else {
            Login newLogin = register.getLogin();
            newLogin = loginRepository.save(newLogin);
            register.setLogin(newLogin);
        }
        return registerRepository.save(register);
    }
    public List<Register> getAllRegisters() {
        Iterable<Register> registers = registerRepository.findAll();
        return StreamSupport.stream(registers.spliterator(),
false).collect(Collectors.toList());
    }
    public Optional<Register> getRegisterById(Long id) {
        return registerRepository.findById(id);
    }
    @Transactional
    public Optional<Register> updateRegister(Long id, Register updatedRegister) {
```

```
return registerRepository.findById(id).map(register -> {
    // Update the Register fields
    register.setName(updatedRegister.getName());
    register.setPhone(updatedRegister.getPhone());
    register.setEmail(updatedRegister.getEmail());
    register.setPassword(updatedRegister.getPassword());
    // Handle the Login entity
    if (register.getLogin() != null) {
        Login existingLogin = register.getLogin();
        existingLogin.setEmail(updatedRegister.getLogin().getEmail());
        existingLogin.setPassword(updatedRegister.getLogin().getPassword());
        loginRepository.save(existingLogin); // Save updated Login
    } else {
        // Handle the case where the Register has no associated Login
        Login newLogin = updatedRegister.getLogin();
        if (newLogin != null) {
            newLogin = loginRepository.save(newLogin); // Save new Login
            register.setLogin(newLogin);
        }
    }
    // Save the updated Register
    return registerRepository.save(register);
});
```

```
    }
    public void deleteAllRegisters() {
        registerRepository.deleteAll();
    }
    @Transactional
    public void deleteLoginById(Long id) {
        Optional<Login> login = loginRepository.findById(id);
        if (login.isPresent()) {
            // Ensure that the login is not referenced by any Register
            Login loginEntity = login.get();
            // Handle any business logic if needed before deleting
            loginRepository.delete(loginEntity);
        }
    }
}
```

5.7. JWTUtil.java

```
package com.example.demo.util;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.Claims;
import java.util.Date;
public class JwtUtil {
    private static final String SECRET_KEY =
"DWnN2JPlmIimWXd3ZJWJtQ9mQOggGynoZpLCtvrGr/M=";
```

```
    private static final long EXPIRATION_TIME = 1000 * 60 * 60; // 1 hour
    public static String generateToken(String username) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() +
EXPIRATION_TIME))
                .signWith(SignatureAlgorithm.HS256, SECRET_KEY)
                .compact();
    }
    public static Claims extractClaims(String token) {
        return Jwts.parserBuilder()
                .setSigningKey(SECRET_KEY)
                .build()
                .parseClaimsJws(token)
                .getBody();
    }
    public static String extractUsername(String token) {
        return extractClaims(token).getSubject();
    }
    public static boolean isTokenExpired(String token) {
        return extractClaims(token).getExpiration().before(new Date());
    }
    public static boolean validateToken(String token, String username) {
        return (username.equals(extractUsername(token)) && !isTokenExpired(token));
    }
}
```

5.8. SecurityConfig.java

```
package com.example.demo.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.web.SecurityFilterChain;
@Configuration
@EnableWebSecurity
public class SecurityConfig {
   @Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
      http
         .csrf(csrf -> csrf.disable())
         .authorizeRequests(auth -> auth
            .requestMatchers("/api/users/register", "/api/users/login",
"/api/applications/**", "/api/job/**").permitAll()  // Allow public access to /api/job/**
            .anyRequest().authenticated());
      return http.build();
   }
}
```

5.9. DataBase

```
spring.application.name=backend

server.port=8080

spring.datasource.url=jdbc:mysql://localhost:3306/backend

spring.datasource.username=root

spring.datasource.password=root
```

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
```

6. Conclusion

The backend of the logistics management website is built with a focus on security, scalability, and efficient data management. By leveraging Spring Boot and related technologies, the application provides a robust and secure platform for managing inventories, transports, and other logistics operations. The integration of JWT for authentication and Spring Security for access control ensures that user data is protected and that only authorized users can access sensitive information. The architecture is designed to be extendable, allowing for future enhancements and scalability as the application grows.

In conclusion, the backend system is well-architected to support the application's requirements, providing a solid foundation for further development and deployment.