Instagram Clone

Complete Project Documentation

Distributed Systems Course

Technical University of Cluj-Napoca

Team Echipa 3

`Distributed Systems 2025`

June 6, 2025

Contents

Chapter 1

Project Overview

1.1 Description

Instagram Clone is a modern web application that replicates the main functionalities of Instagram, built with a microservices architecture using Spring Boot for the backend and React for the frontend.

1.2 Key Features

- **User Management**: Registration, authentication, user profiles
- **Post Management**: Create, edit, delete posts with images
- **Comment System**: Comments on posts with images
- **Reaction System**: Like/Dislike for posts and comments
- **Tag System**: Organizing posts with hashtags
- **Admin Panel**: User and content management
- **Real-time Feed**: Feed with chronologically sorted posts
- **User Profiles**: Profile pages with statistics and posts

1.3 Technology Stack

1.3.1 Backend

- Java 8
- Spring Boot 2.3.3
- Spring Data JPA
- Spring HATEOAS
- MySQL 8.0

- Maven
- JaCoCo (Code Coverage)
- Checkstyle

1.3.2 Frontend

- React 16.8
- React Router 5.3.4
- Reactstrap & Bootstrap 5
- Axios
- Context API
- SCSS/CSS

1.3.3 DevOps & Tools

- GitLab CI/CD
- Heroku
- H2 Database (testing)
- Cypress (E2E testing)

Chapter 2

System Architecture

2.1 Microservices Architecture

The application follows a microservices architecture pattern with three main services:

Service	**Port**	**Purpose**
People API	8080	User management and authentication
Posts API	8081	Content management system
Reactions API	8082	User engagement tracking
React Frontend	3000	User interface

Table 2.1: Microservices Overview

2.2 Service Communication

- **Synchronous Communication**: HTTP REST API calls
- **Database Synchronization**: Cross-service cleanup operations
- **Frontend Integration**: Axios-based API clients

2.3 Database Design

Each microservice has its own dedicated database:

- `instagram-people`: User data
- `instagram-posts`: Posts, comments, and tags
- `instagram-reactions`: User reactions

Chapter 3

Backend Microservices

3.1 People Microservice (Port 8080)

3.1.1 Purpose

User management and authentication system.

3.1.2 Entities

- **Person**: User profiles and authentication data

3.1.3 Key Features

- User registration and authentication
- Profile management (CRUD operations)
- Admin user management
- Role-based access control

3.1.4 API Endpoints

Method	**Endpoint**	**Description**
GET	/people	Get all users
POST	/people	Create user
GET	/people/{id}	Get user by ID
POST	/people/{id}	Update user
DELETE	/people/{id}	Delete user
GET	/people/authenticate/{username}/{password}	User authentication
GET	/people/admins	Get admin users

Table 3.1: People API Endpoints

3.1.5 Database Schema

```sql
CREATE TABLE person (
    id_person INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    username VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    user_score INT NOT NULL DEFAULT 0,
    is_admin BOOLEAN NOT NULL DEFAULT FALSE,
    is_banned BOOLEAN NOT NULL DEFAULT FALSE,
    email VARCHAR(255) NOT NULL,
    phone_number VARCHAR(20) NOT NULL,
    birth_date DATETIME,
    home_city VARCHAR(255)
);
```

Listing 3.1: Person Table Schema

3.2 Posts Microservice (Port 8081)

3.2.1 Purpose

Content management system for posts, comments, and tags.

3.2.2 Entities

- **Post**: Posts and comments
- **Tag**: Hashtags for organizing content
- **PostTag**: Many-to-many relationship between posts and tags

3.2.3 Key Features

- Post creation with images
- Comment system
- Tag management
- Post status tracking
- Content organization

3.2.4 API Endpoints

Method	Endpoint	Description
Posts		
GET	/posts	Get all posts
POST	/posts	Create post
GET	/posts/{id}	Get post by ID

POST	/posts/{id}	Update post
DELETE	/posts/{id}	Delete post
DELETE	/posts/person/{id}	Delete all posts from user
Tags		
GET	/tags	Get all tags
POST	/tags	Create tag
GET	/tags/{id}	Get tag by ID
GET	/tags/name/{name}	Get tag by name
POST	/tags/{id}	Update tag
DELETE	/tags/{id}	Delete tag
PostTags		
GET	/postTags	Get all post-tag relations
POST	/postTags	Create post-tag relation
GET	/postTags/{id}	Get post-tag by ID
GET	/postTags/post/{id}	Get tags for post
POST	/postTags/{id}	Update post-tag
DELETE	/postTags/{id}	Delete post-tag

Table 3.2: Posts API Endpoints

3.2.5 Database Schema

```sql
CREATE TABLE post (
    id_post INT PRIMARY KEY AUTO_INCREMENT,
    id_person INT NOT NULL,
    id_parent INT,                    -- For comments
    id_tag INT,
    title VARCHAR(255) NOT NULL,
    text TEXT NOT NULL,
    date_created DATETIME NOT NULL,
    status VARCHAR(50) NOT NULL,
    image LONGBLOB,
    total_votes INT DEFAULT 0,
    no_more_comments BOOLEAN DEFAULT FALSE
);

CREATE TABLE tag (
    id_tag INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) UNIQUE NOT NULL
);

CREATE TABLE post_tag (
    id_post_tag INT PRIMARY KEY AUTO_INCREMENT,
    id_post INT NOT NULL,
    id_tag INT NOT NULL,
    FOREIGN KEY (id_post) REFERENCES post(id_post),
    FOREIGN KEY (id_tag) REFERENCES tag(id_tag)
);
```

Listing 3.2: Posts Database Schema

3.3 Reactions Microservice (Port 8082)

3.3.1 Purpose

User engagement tracking through likes and dislikes.

3.3.2 Entities

- **Reaction**: Like/Dislike reactions on posts and comments

3.3.3 Key Features

- Like/Dislike functionality
- Reaction tracking
- Vote counting
- User engagement analytics

3.3.4 API Endpoints

Method	**Endpoint**	**Description**
GET	/reactions	Get all reactions
POST	/reactions	Create reaction
GET	/reactions/{id}	Get reaction by ID
POST	/reactions/{id}	Update reaction
DELETE	/reactions/{id}	Delete reaction
DELETE	/reactions/post/{id}	Delete all reactions from post
DELETE	/reactions/person/{id}	Delete all reactions from user

Table 3.3: Reactions API Endpoints

3.3.5 Database Schema

```sql
CREATE TABLE reaction (
    id_reaction INT PRIMARY KEY AUTO_INCREMENT,
    id_person INT NOT NULL,
    id_post INT NOT NULL,
    is_liked BOOLEAN NOT NULL
);
```

Listing 3.3: Reactions Database Schema

Chapter 4

Frontend Application

4.1 Technology Stack

- **React 16.8** with Hooks and Class Components
- **React Router 5.3.4** for navigation
- **Context API** for state management
- **Reactstrap & Bootstrap 5** for UI components
- **Axios** for HTTP requests

4.2 Application Structure

```
src/
├─ admin/
│  └─ Admin panel
├─ api/
│  └─ API clients
├─ components/
│  └─ Admin components
├─ admin-container.js
│  └─ Main admin page
├─ commons/
│  └─ Shared components
├─ api/
│  └─ Base API client
├─ errorhandling/
│  └─ Error components
├─ images/
│  └─ Static images
├─ styles/
│  └─ CSS styles
├─ tables/
│  └─ Table components
├─ contexts/
│  └─ React contexts
├─ UserContext.js
│  └─ User state management
├─ login/
│  └─ Authentication
├─ components/
│  └─ Login components
├─ login.js
│  └─ Main login page
├─ person/
```

4.3 Key Components

4.3.1 Authentication System

- **Login/Register**: User authentication with validation
- **UserContext**: Global user state management
- **Protected Routes**: Role-based access control

4.3.2 Feed System

- **Feed**: Main timeline with posts
- **Post Cards**: Individual post display
- **Infinite Scroll**: Performance optimization
- **Real-time Updates**: Live content updates

4.3.3 User Profiles

- **Profile Display**: User information and statistics
- **Post Grid**: User's posts in grid layout
- **Edit Profile**: Profile management interface

4.3.4 Content Management

- **Post Creation**: Multi-step post creation with image upload
- **Comment System**: Nested comments with reactions
- **Tag System**: Hashtag creation and organization

4.3.5 Admin Panel

- **User Management**: CRUD operations for users
- **Content Moderation**: Post and comment management
- **Tag Administration**: Tag system management

4.4 State Management

4.4.1 UserContext Structure

```
const UserContext = createContext();

// User state structure
{
  idPerson: number,
  name: string,
  username: string,
  userScore: number,
  isAdmin: boolean,
  isBanned: boolean,
  email: string,
  phoneNumber: string,
  birthDate: string,
  homeCity: string
}
```

Listing 4.1: User Context Structure

4.4.2 API Integration

```
// Example API client structure
const endpoint = {
    posts: '/posts'
};

function getPosts(callback) {
    let request = new Request(HOST.posts_api + endpoint.posts, {
        method: 'GET',
    });
    RestApiClient.performRequest(request, callback);
}
```

Listing 4.2: API Client Example

Chapter 5

Database Design

5.1 Entity Relationships

The database design follows a normalized relational model with the following key relationships:

- **Person** → **Post**: One-to-many (one user creates many posts)
- **Person** → **Reaction**: One-to-many (one user makes many reactions)
- **Post** → **Post**: One-to-many (post has comments)
- **Post** → **Reaction**: One-to-many (post receives many reactions)
- **Post** ↔ **Tag**: Many-to-many (through PostTag)

5.2 Database Configuration

5.2.1 Development Environment

```
# MySQL Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/instagram-{service}
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=validate
```

Listing 5.1: MySQL Configuration

5.2.2 Test Environment

```
# H2 In-Memory Database
spring.datasource.url=jdbc:h2:mem:jpa_jbd
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=create-drop
```

Listing 5.2: H2 Test Configuration

Chapter 6

API Documentation

6.1 Authentication

All endpoints support cross-origin requests using the `@CrossOrigin` annotation.

6.2 Response Format

6.2.1 Success Response

```
{
  "data": "any",
  "status": "200-299"
}
```

Listing 6.1: Success Response Format

6.2.2 Error Response

```
{
  "timestamp": "2025-01-01T10:00:00",
  "status": "400-500",
  "error": "Error Type",
  "message": "Error Description",
  "path": "/api/endpoint",
  "details": []
}
```

Listing 6.2: Error Response Format

6.3 Common HTTP Status Codes

6.4 API Client Configuration

```
// Host configuration
const HOST = {
    people_api: 'http://localhost:8080',
    posts_api: 'http://localhost:8081',
```

Status Code	**Description**
200 OK	Successful GET/PUT operations
201 Created	Successful POST operations
202 Accepted	Successful DELETE operations
400 Bad Request	Invalid input data
404 Not Found	Resource not found
409 Conflict	Duplicate resource
422 Unprocessable Entity	Validation errors

Table 6.1: HTTP Status Codes

```
    reactions_api: 'http://localhost:8082'
};
```

Listing 6.3: Host Configuration

Chapter 7

Installation & Setup

7.1 Prerequisites

- **Java 8+**
- **Node.js 14+**
- **MySQL 8.0+**
- **Maven 3.6+**
- **Git**

7.2 Backend Setup

7.2.1 Clone Repository

```
git clone <repository-url>
cd Instagram
```

Listing 7.1: Repository Clone

7.2.2 Database Setup

```
-- Create databases for each microservice
CREATE DATABASE `instagram-people`;
CREATE DATABASE `instagram-posts`;
CREATE DATABASE `instagram-reactions`;

-- Create user with permissions
CREATE USER 'instagram'@'localhost' IDENTIFIED BY 'password';
GRANT ALL PRIVILEGES ON `instagram-%`.* TO 'instagram'@'localhost';
FLUSH PRIVILEGES;
```

Listing 7.2: Database Creation

7.2.3 Configure Application Properties

```bash
# Set environment variables or update application.properties
export DB_IP=localhost
export DB_PORT=3306
export DB_USER=instagram
export DB_PASSWORD=password
```

Listing 7.3: Environment Variables

7.2.4 Build and Run Microservices

```bash
# People Service
cd Backend/People
mvn clean install
mvn spring-boot:run

# Posts Service
cd ../Posts
mvn clean install
mvn spring-boot:run

# Reactions Service
cd ../Reactions
mvn clean install
mvn spring-boot:run
```

Listing 7.4: Microservice Startup

7.3 Frontend Setup

7.3.1 Install Dependencies

```bash
cd Frontend/react-demo
npm install
```

Listing 7.5: Frontend Dependencies

7.3.2 Configure API Endpoints

```javascript
// src/commons/hosts.js
export const HOST = {
    people_api: 'http://localhost:8080',
    posts_api: 'http://localhost:8081',
    reactions_api: 'http://localhost:8082'
};
```

Listing 7.6: API Configuration

7.3.3 Start Development Server

```
npm start
# Application will be available at http://localhost:3000
```

Listing 7.7: Frontend Startup

7.4 Docker Setup (Optional)

7.4.1 Dockerfile Example

```
# Dockerfile example for backend service
FROM openjdk:8-jre-slim
COPY target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Listing 7.8: Backend Dockerfile

7.4.2 Docker Compose

```
# docker-compose.yml
version: '3.8'
services:
  mysql:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: root
    ports:
      - "3306:3306"

  people-service:
    build: ./Backend/People
    ports:
      - "8080:8080"
    depends_on:
      - mysql

  posts-service:
    build: ./Backend/Posts
    ports:
      - "8081:8081"
    depends_on:
      - mysql

  reactions-service:
    build: ./Backend/Reactions
    ports:
      - "8082:8082"
    depends_on:
      - mysql

  frontend:
    build: ./Frontend/react-demo
    ports:
      - "3000:3000"
    depends_on:
      - people-service
      - posts-service
      - reactions-service
```

Listing 7.9: Docker Compose Configuration

Chapter 8

User Guide

8.1 Getting Started

8.1.1 Registration

1. Visit the application homepage
2. Click ”Sign up” button
3. Fill in registration form:
    - Name, Username, Password
    - Email, Phone Number
    - Birth Date, Home City
4. Submit form and wait for confirmation
5. Use your credentials to log in

8.1.2 Navigation

- **Home/Feed**: View all posts from users
- **Profile**: Manage your profile and view your posts
- **Admin** (admin only): User and content management

8.2 Core Features

8.2.1 Creating Posts

1. Navigate to Home/Feed
2. Click ”New Post” button
3. Fill in post details:
    - Title (required)

- Description (required)
- Tags (space-separated, with # prefix)
- Image (optional)

4. Click ”Submit” to publish

8.2.2 Interacting with Posts

- **View Details**: Click on any post card
- **Like/Dislike**: Use reaction buttons (not on your own posts)
- **Comment**: Add comments with images (if enabled)
- **Edit/Delete**: Manage your own posts

8.2.3 Profile Management

1. Navigate to Profile page
2. View your statistics and posts
3. Click ”Edit Profile” to update information
4. Save changes to update your profile

8.2.4 Admin Functions (Admin Users Only)

1. Navigate to Admin panel
2. **User Management**:
   - Add new users
   - Edit user information
   - Delete users
   - View user statistics
3. **Tag Management**:
   - Create new tags
   - Delete unused tags

8.3 Post Status Lifecycle

1. **”Just Posted”**: Newly created posts
2. **”First Reactions”**: Posts with at least one comment
3. **”Outdated”**: Posts with disabled comments

Chapter 9

Development Guide

9.1 Code Standards

9.1.1 Backend (Java)

- **Checkstyle**: Enforced code style rules
- **JavaDoc**: Comprehensive code documentation
- **SOLID Principles**: Clean architecture patterns
- **REST API Standards**: Consistent endpoint naming

9.1.2 Frontend (JavaScript/React)

- **ESLint**: Code quality enforcement
- **Component Structure**: Functional and class components
- **State Management**: Context API best practices
- **Responsive Design**: Mobile-first approach

9.2 Development Workflow

9.2.1 Feature Development

```
# Create feature branch
git checkout -b feature/new-feature

# Make changes and commit
git add .
git commit -m "feat: add new feature"

# Push and create merge request
git push origin feature/new-feature
```

Listing 9.1: Feature Development Workflow

9.2.2 Testing Strategy

```
# Backend testing
mvn test                        # Unit tests
mvn verify                      # Integration tests
mvn jacoco:report              # Coverage report

# Frontend testing
npm test                       # Unit tests
npm run cypress:open           # E2E tests
```

Listing 9.2: Testing Commands

9.2.3 Code Quality Gates

- **Unit Test Coverage**: Minimum 80%
- **Integration Tests**: All API endpoints
- **Code Style**: Zero checkstyle violations
- **Security**: Input validation and sanitization

9.3 Adding New Features

9.3.1 Backend - New Endpoint

1. **Create Entity**: Define JPA entity
2. **Create Repository**: Extend JpaRepository
3. **Create Service**: Business logic layer
4. **Create Controller**: REST endpoint
5. **Add Tests**: Unit and integration tests
6. **Update Documentation**: API docs

9.3.2 Frontend - New Component

1. **Create Component**: React component file
2. **Add Styling**: CSS/SCSS styles
3. **Integrate API**: Connect to backend
4. **Add Navigation**: Router integration
5. **Add Tests**: Component tests
6. **Update UI**: Navigation and links

9.4 Database Migration

```sql
-- Example migration script
ALTER TABLE person ADD COLUMN profile_image LONGBLOB;
UPDATE person SET profile_image = NULL WHERE profile_image IS NULL;
```

Listing 9.3: Database Migration Example

Chapter 10

Testing

10.1 Backend Testing

10.1.1 Unit Tests

```java
@Test
public void testInsertCorrectWithGetById() {
    PersonDTO person = new PersonDTO(/* parameters */);
    Integer insertedID = personService.insert(person);
    PersonDTO fetchedPerson = personService.findPersonById(insertedID);
    assertEquals("Test Inserted Person", insertedPerson, fetchedPerson);
}
```

Listing 10.1: Unit Test Example

10.1.2 Integration Tests

```java
@Sql(executionPhase = Sql.ExecutionPhase.BEFORE_TEST_METHOD,
     scripts = "classpath:/test-sql/create.sql")
@Sql(executionPhase = Sql.ExecutionPhase.AFTER_TEST_METHOD,
     scripts = "classpath:/test-sql/delete.sql")
public class PersonServiceIntegrationTests extends Ds2020TestConfig {
    // Test methods
}
```

Listing 10.2: Integration Test Example

10.1.3 Test Configuration

```
# Test database configuration
spring.datasource.url=jdbc:h2:mem:jpa_jbd
spring.jpa.hibernate.ddl-auto=create-drop
```

Listing 10.3: Test Database Configuration

10.2 Frontend Testing

10.2.1 Component Tests

```
import { render, screen } from '@testing-library/react';
import PersonContainer from './PersonContainer';

test('renders user profile', () => {
  render(<PersonContainer />);
  const linkElement = screen.getByText(/profile/i);
  expect(linkElement).toBeInTheDocument();
});
```

Listing 10.4: Component Test Example

10.2.2 E2E Tests (Cypress)

```
describe('Instagram Clone', () => {
  it('should login and create post', () => {
    cy.visit('/');
    cy.get('[data-cy=username]').type('testuser');
    cy.get('[data-cy=password]').type('password');
    cy.get('[data-cy=login-btn]').click();
    cy.url().should('include', '/home');
  });
});
```

Listing 10.5: Cypress E2E Test

10.3 Test Coverage Goals

Component	**Coverage Target**
Backend	¿80% line coverage
Frontend	¿70% component coverage
E2E	Critical user flows
API	All endpoints tested

Table 10.1: Test Coverage Goals

Chapter 11

Deployment

11.1 Production Environment

11.1.1 Backend Deployment (Heroku)

```yaml
# .gitlab-ci.yml
stages:
  - build
  - deploy

build:
  stage: build
  script:
    - mvn clean install

deploy:
  stage: deploy
  script:
    - git push heroku main
  only:
    - production
```

Listing 11.1: GitLab CI/CD Configuration

11.1.2 Frontend Deployment

```yaml
# Frontend deployment
build:
  stage: build
  image: node:11
  script:
    - npm install --progress=false
    - npm run build

deploy:
  stage: deploy
  script:
    - dpl --provider=heroku --app=instagram-frontend --api-key=
   $HEROKU_API_KEY
```

Listing 11.2: Frontend Deployment Configuration

11.1.3 Environment Variables

```
# Production environment variables
DB_IP=production-db-host
DB_PORT=3306
DB_USER=prod_user
DB_PASSWORD=secure_password
JWT_SECRET=secure_jwt_secret
```

Listing 11.3: Production Environment Variables

11.1.4 Health Checks

```
// Spring Boot Actuator endpoints
/actuator/health     // Service health
/actuator/metrics    // Performance metrics
/actuator/info       // Application information
```

Listing 11.4: Spring Boot Actuator Endpoints

11.2 Monitoring & Logging

```
# Logging configuration
logging.level.ro.tuc=INFO
logging.pattern.file=%d{yyyy-MM-dd HH:mm:ss} - %msg%n
logging.file.name=app.log
```

Listing 11.5: Logging Configuration

Chapter 12

Troubleshooting

12.1 Common Issues

12.1.1 Backend Issues

Database Connection Failed

```
# Check MySQL service
sudo systemctl status mysql

# Verify credentials
mysql -u root -p

# Check application.properties
spring.datasource.url=jdbc:mysql://localhost:3306/instagram-people
```

Listing 12.1: Database Troubleshooting

Port Already in Use

```
# Find process using port
lsof -i :8080

# Kill process
kill -9 <PID>

# Or change port in application.properties
server.port=8083
```

Listing 12.2: Port Conflict Resolution

JPA/Hibernate Errors

```
# Enable SQL logging
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

# Check DDL mode
spring.jpa.hibernate.ddl-auto=validate
```

Listing 12.3: JPA Debug Configuration

12.1.2 Frontend Issues

API Connection Failed

```
// Check API hosts configuration
export const HOST = {
    people_api: 'http://localhost:8080',
    posts_api: 'http://localhost:8081',
    reactions_api: 'http://localhost:8082'
};
```

Listing 12.4: API Configuration Check

CORS Issues

```
// Ensure @CrossOrigin is present
@RestController
@CrossOrigin
public class PersonController {
    // Controller methods
}
```

Listing 12.5: CORS Configuration

Build Failures

```
# Clear npm cache
npm cache clean --force

# Delete node_modules and reinstall
rm -rf node_modules package-lock.json
npm install

# Check Node.js version
node --version  # Should be 14+
```

Listing 12.6: Frontend Build Issues

12.2 Performance Issues

12.2.1 Slow Database Queries

```
-- Add indexes for frequently queried columns
CREATE INDEX idx_person_username ON person(username);
CREATE INDEX idx_post_person ON post(id_person);
CREATE INDEX idx_post_date ON post(date_created);
```

Listing 12.7: Database Optimization

12.2.2 Memory Issues

```
# Increase JVM memory
java -Xmx2g -Xms1g -jar app.jar

# Or in application.properties
spring.jpa.open-in-view=false
```

Listing 12.8: Memory Configuration

12.2.3 Frontend Performance

```
// Lazy loading for routes
const LazyComponent = React.lazy(() => import('./Component'));

// Memoization for expensive calculations
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b])
    ;
```

Listing 12.9: Performance Optimization

12.3 Debug Mode

12.3.1 Backend Debug

```
# Run with debug mode
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005
    -jar app.jar
```

Listing 12.10: Backend Debug Mode

12.3.2 Frontend Debug

```
// Enable React DevTools
npm install -g react-devtools

// Debug mode
NODE_ENV=development npm start
```

Listing 12.11: Frontend Debug Mode

Chapter 13

License & Credits

13.1 License

This project is developed for educational purposes as part of the Distributed Systems course at Technical University of Cluj-Napoca.

13.2 Contributors

- **Backend Development**: Spring Boot microservices architecture
- **Frontend Development**: React application with modern UI/UX
- **Database Design**: Normalized relational database schema
- **DevOps**: CI/CD pipeline and deployment automation

13.3 Third-Party Libraries

13.3.1 Backend

- Spring Boot Starter Pack
- MySQL Connector
- JaCoCo (Code Coverage)
- JUnit (Testing)
- Hibernate Validator

13.3.2 Frontend

- React & React DOM
- React Router
- Reactstrap & Bootstrap
- Axios (HTTP Client)

- Pacifico Font
- Cypress (E2E Testing)

Chapter 14

Support & Contact

14.1 Getting Help

1. **Documentation**: Refer to this comprehensive guide
2. **Issues**: Create GitHub/GitLab issues for bugs
3. **Discussions**: Use project discussion boards
4. **Code Review**: Submit merge requests for contributions

14.2 Development Team

- **Architecture**: Microservices design and implementation
- **Backend**: RESTful API development
- **Frontend**: React application development
- **Database**: Schema design and optimization
- **DevOps**: Deployment and CI/CD pipeline

Technical University of Cluj-Napoca