**Project Documentation: Social Media Platform with Payment Integration**

**1. Introduction**

This project is a social media platform enabling users to create profiles, post content, and upgrade to a "Pro" tier via PayPal. Administrators can manage users and content. The platform is built using Java Spring Boot, PostgreSQL, and Thymeleaf, ensuring scalability and maintainability.

**2. Tech Stack**

| **Category** | **Tools** |
| --- | --- |
| Backend Framework | Java Spring Boot (Maven) |
| Frontend | Thymeleaf, HTML/CSS/JavaScript, Bootstrap |
| Database | H2 (development), PostgreSQL (production) |
| Security | Spring Security, BCryptPasswordEncoder |
| Payment Gateway | PayPal Sandbox API |
| Email Service | SendGrid (Free Tier) |
| Deployment | Heroku (Free Tier) or Render |
| Tools | Postman (API testing), GitHub (version control), IntelliJ/Eclipse (IDE) |

**3. System Architecture**

*Diagram Placeholder: Illustrate the flow between frontend (Thymeleaf), backend (Spring Boot), database (PostgreSQL), and external services (PayPal, SendGrid). Use tools like draw.io to create.*

The architecture follows a layered MVC pattern:

* **Frontend**: Thymeleaf templates render dynamic views.
* **Backend**: Spring Boot handles business logic and APIs.
* **Database**: PostgreSQL stores user data, posts, and payments.
* **External Services**: PayPal for payments, SendGrid for emails.

**4. Features**

**4.1 User Roles**

* **Admin**: Manage users (block/delete), edit/delete posts, view contact messages.
* **Customer**: Create/edit posts, update profile, contact support, upgrade to Pro tier.

**4.2 Core Features**

* User registration/login with password visibility toggle.
* Dashboard displaying all posts (filterable by user).
* Profile page for editing username, email, and bio.
* Contact form integrated with SendGrid for support queries.
* PayPal payment integration for Pro tier upgrades.
* Role-based access control using Spring Security.

**5. Setup Guide**

**5.1 Prerequisites**

* Java JDK 17 or higher
* Maven 3.8 or higher
* PostgreSQL 13+ (for production)
* PayPal Developer Account (Sandbox)
* SendGrid account (Free Tier)

**5.2 Local Development Setup**

1. Clone the repository:

bash

Copy

git clone https://github.com/yourusername/social-media-platform.git

1. Navigate to the project directory:

bash

Copy

cd social-media-platform

1. Configure src/main/resources/application.properties for H2 database:

properties

Copy

spring.datasource.url=jdbc:h2:mem:socialdb

spring.datasource.username=sa

spring.datasource.password=

spring.h2.console.enabled=true

spring.jpa.hibernate.ddl-auto=update

1. Install dependencies and run:

bash

Copy

mvn clean install

mvn spring-boot:run

1. Access the app at http://localhost:8080 and H2 console at http://localhost:8080/h2-console.

**6. Database Design**

**6.1 Entity Relationship Diagram (ERD)**

*Diagram Placeholder: Show relationships between Users, Posts, Payments, and Messages tables. Create using draw.io or Lucidchart.*

**6.2 SQL Schema**

sql

Copy

CREATE TABLE users (

id SERIAL PRIMARY KEY,

username VARCHAR(50) UNIQUE NOT NULL,

password VARCHAR(100) NOT NULL,

email VARCHAR(100) UNIQUE NOT NULL,

role VARCHAR(10) DEFAULT 'USER',

is\_pro BOOLEAN DEFAULT false,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE posts (

id SERIAL PRIMARY KEY,

content TEXT NOT NULL,

user\_id INT REFERENCES users(id) ON DELETE CASCADE,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE payments (

id SERIAL PRIMARY KEY,

user\_id INT REFERENCES users(id) ON DELETE CASCADE,

paypal\_order\_id VARCHAR(100),

amount DECIMAL(10,2),

status VARCHAR(20),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE messages (

id SERIAL PRIMARY KEY,

user\_id INT REFERENCES users(id) ON DELETE SET NULL,

subject VARCHAR(100),

content TEXT,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

**7. Deployment**

**7.1 Heroku Deployment**

1. Install Heroku CLI and log in:

bash

Copy

heroku login

1. Create a Procfile in the project root:

Procfile

Copy

web: java -jar target/\*.jar --server.port=$PORT

1. Deploy to Heroku:

bash

Copy

heroku create your-app-name

git push heroku main

heroku addons:create heroku-postgresql:hobby-dev

1. Set environment variables (e.g., PayPal credentials) using:

bash

Copy

heroku config:set KEY=VALUE

**7.2 Configure Production Database**

Update application.properties for PostgreSQL:

properties

Copy

spring.datasource.url=${JDBC\_DATABASE\_URL}

spring.datasource.username=${JDBC\_DATABASE\_USERNAME}

spring.datasource.password=${JDBC\_DATABASE\_PASSWORD}

spring.jpa.hibernate.ddl-auto=update

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

**8. Security**

* **Password Hashing**: BCryptPasswordEncoder for secure password storage.
* **HTTPS**: Enforced in production via Spring Security configuration.
* **CSRF Protection**: Enabled by default in Spring Security.
* **Role-Based Access**: Admins access /admin/\* endpoints; users access /user/\*.

**9. Payment Integration (PayPal)**

1. Create a PayPal Sandbox account at [developer.paypal.com](https://developer.paypal.com/).
2. Add PayPal SDK to pom.xml:

xml

Copy

<dependency>

<groupId>com.paypal.sdk</groupId>

<artifactId>paypal-core</artifactId>

<version>1.9.1</version>

</dependency>

1. Configure PayPal credentials in application.properties:

properties

Copy

paypal.client.id=${PAYPAL\_CLIENT\_ID}

paypal.client.secret=${PAYPAL\_CLIENT\_SECRET}

paypal.mode=sandbox

1. Implement payment controller:

java

Copy

@RestController

@RequestMapping("/api/payment")

public class PaymentController {

@Autowired

private PayPalService payPalService;

@PostMapping("/create")

public String createPayment(@RequestParam Double amount) {

*// Logic to create PayPal payment*

return payPalService.createPayment(amount, "USD", "Pro Upgrade");

}

@GetMapping("/success")

public String paymentSuccess(@RequestParam("paymentId") String paymentId, @RequestParam("PayerID") String payerId) {

*// Complete payment and update user to Pro*

return "redirect:/user/dashboard";

}

}

**10. Testing**

* **Unit Tests**: Use JUnit and Mockito for backend services.
* **API Testing**: Use Postman to test endpoints:
  + POST /api/auth/register
  + POST /api/posts
  + POST /api/payment/create
* **Manual Testing**: Validate registration, post creation, payment flow, and admin permissions.
* **Test Cases**:
  + User can register and log in.
  + Pro users access premium features.
  + Admins can delete posts/users.

**11. Future Enhancements**

* Real-time chat using WebSocket or STOMP.
* File uploads (images/videos) for posts using AWS S3.
* OAuth2 authentication (Google, Facebook).
* Mobile app integration with REST APIs.

**12. Conclusion**

This document provides a comprehensive guide to the platform’s setup, architecture, and deployment. For issues or contributions, visit the [GitHub repository](https://github.com/yourusername/social-media-platform) or contact support via the platform’s contact form.

**Appendices**

* **Postman Collection**: [Placeholder: Export and share your Postman collection]
* **GitHub Repository**: [Placeholder: Link to actual repo]

**How to Convert to DOCX**

1. Copy the entire markdown content above.
2. Open Microsoft Word or Google Docs.
3. Paste the content.
4. Format headings (e.g., Heading 1 for #, Heading 2 for ##).
5. Convert tables to Word tables (use "Insert Table" or Google Docs’ table tool).
6. Style code blocks with a monospaced font (e.g., Consolas) and a gray background.
7. Replace placeholder image links with actual diagrams:
   * Use [draw.io](https://draw.io) or Lucidchart to create architecture and ERD diagrams.
   * Export diagrams as PNG/JPG and insert into the document.
8. Save as .docx (File > Save As > Word Document).

**Additional Help**

* If you need specific diagrams, I can describe them in detail (e.g., architecture flow, ERD relationships).
* If you want code snippets expanded (e.g., full PayPalService class), let me know.
* For GitHub repo setup or Postman collection creation, I can guide you step-by-step.

**Project Roadmap: Social Media Platform with Payment Integration**

**Tech Stack:**

* **Backend:** Java Spring Boot (v3.2+), Maven
* **Frontend:** Thymeleaf, HTML5/CSS3/JavaScript, Bootstrap 5
* **Database:** H2 (development), PostgreSQL 15+ (production)
* **Security:** Spring Security, BCryptPasswordEncoder
* **Payment:** PayPal REST API (Sandbox for testing)
* **Email:** SendGrid (free tier, 100 emails/day)
* **Deployment:** Heroku (free tier), Render, or AWS Free Tier (EC2/Lightsail)
* **Tools:** GitHub (version control), Postman (API testing), IntelliJ IDEA or Eclipse (IDE)

**Timeline Overview:**

* Total Duration: ~8-10 weeks (assuming part-time development).
* Each phase includes development, testing, and documentation.

**Phase 1: Setup & Configuration**

**Duration**: 1 week  
**Goals**: Establish project foundation, configure development environment.

1. **Initialize Project**
   * Use [Spring Initializr](https://start.spring.io/) to generate a Spring Boot project:
     + **Dependencies**: Spring Web, Spring Data JPA, Spring Security, Thymeleaf, H2 Database, PostgreSQL Driver, Lombok, JavaMail Sender, PayPal SDK (e.g., com.paypal.sdk:rest-api-sdk).
     + **Java Version**: 17.
     + **Build Tool**: Maven.
   * Create GitHub repository with .gitignore for Spring Boot.
2. **Project Structure**

text

Copy

src/main/

├── java/com/yourdomain/

│ ├── config/ (SecurityConfig, PayPalConfig)

│ ├── controller/ (AuthController, PostController, etc.)

│ ├── model/ (User, Post, Payment, Message)

│ ├── repository/ (UserRepository, PostRepository, etc.)

│ ├── service/ (UserService, PaymentService, EmailService)

│ └── Application.java

└── resources/

├── static/ (css/, js/, images/)

├── templates/ (home.html, login.html, etc.)

├── application.properties

└── application-prod.properties

1. **Database Setup**
   * Configure application.properties for H2 (development):

properties

Copy

spring.datasource.url=jdbc:h2:mem:socialdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.h2.console.enabled=true

spring.jpa.hibernate.ddl-auto=update

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect

* + Test H2 console at http://localhost:8080/h2-console.

1. **Environment Setup**
   * Install Java 17, Maven 3.8+, and PostgreSQL locally.
   * Configure IDE (IntelliJ/Eclipse) with Lombok plugin.
   * Set up Postman for API testing.

**Deliverables**:

* Running Spring Boot app (mvn spring-boot:run).
* GitHub repo with initial commit.
* H2 database accessible locally.

**Phase 2: User Authentication & Roles**

**Duration**: 1.5 weeks  
**Goals**: Implement secure authentication and role-based access.

1. **User Entity**

java

Copy

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(unique = true)

private String username;

private String password;

@Column(unique = true)

private String email;

private String role; *// "ADMIN" or "USER"*

private boolean isPro; *// Tracks Pro tier status*

private LocalDateTime createdAt;

*// Getters, setters (use Lombok @Data if preferred)*

}

1. **Spring Security Configuration**
   * Create SecurityConfig.java:

java

Copy

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(auth -> auth

.requestMatchers("/admin/\*\*").hasRole("ADMIN")

.requestMatchers("/user/\*\*").hasAnyRole("USER", "ADMIN")

.requestMatchers("/", "/login", "/register").permitAll()

.anyRequest().authenticated()

)

.formLogin(form -> form

.loginPage("/login")

.defaultSuccessUrl("/dashboard")

.permitAll()

)

.logout(logout -> logout

.logoutUrl("/logout")

.logoutSuccessUrl("/login")

.permitAll()

);

return http.build();

}

}

1. **Registration & Login Pages**
   * Create Thymeleaf templates:
     + login.html: Custom form with username/password fields.
     + register.html: Form for username, email, password.
     + error.html: Handle login/registration errors.
   * Implement AuthController:

java

Copy

@Controller

public class AuthController {

@Autowired

private UserService userService;

@GetMapping("/register")

public String showRegisterForm(Model model) {

model.addAttribute("user", new User());

return "register";

}

@PostMapping("/register")

public String registerUser(@ModelAttribute User user) {

userService.registerUser(user);

return "redirect:/login";

}

}

**Deliverables**:

* User registration/login working locally.
* Role-based access (e.g., /admin redirects non-admins).
* Passwords hashed with BCrypt.

**Phase 3: User Profile, Posts & Dashboard**

**Duration**: 2 weeks  
**Goals**: Enable users to manage profiles and create/view posts.

1. **Profile Page**
   * Create profile.html:
     + Form to edit username, email, bio (optional).
     + Display Pro status and upgrade option.
   * Implement UserController:

java

Copy

@Controller

@RequestMapping("/user")

public class UserController {

@GetMapping("/profile")

public String showProfile(Model model, Authentication auth) {

User user = userService.findByUsername(auth.getName());

model.addAttribute("user", user);

return "profile";

}

@PostMapping("/profile")

public String updateProfile(@ModelAttribute User user) {

userService.updateUser(user);

return "redirect:/user/profile";

}

}

1. **Post Entity & Dashboard**

java

Copy

@Entity

public class Post {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(columnDefinition = "TEXT")

private String content;

@ManyToOne

@JoinColumn(name = "user\_id")

private User author;

private LocalDateTime createdAt;

*// Getters, setters*

}

* + Create PostController:

java

Copy

@Controller

public class PostController {

@Autowired

private PostService postService;

@GetMapping("/dashboard")

public String showDashboard(Model model) {

model.addAttribute("posts", postService.findAllPosts());

return "dashboard";

}

@PostMapping("/posts")

public String createPost(@RequestParam String content, Authentication auth) {

postService.createPost(content, auth.getName());

return "redirect:/dashboard";

}

}

* + Create dashboard.html:
    - Display posts in a card layout (Bootstrap).
    - Include a form for creating new posts.

1. **Admin Features** *(Added Content)*:
   * Create admin.html:
     + List all users and posts.
     + Buttons to delete users/posts or toggle Pro status.
   * Implement AdminController:

java

Copy

@Controller

@RequestMapping("/admin")

public class AdminController {

@GetMapping("/dashboard")

public String adminDashboard(Model model) {

model.addAttribute("users", userService.findAllUsers());

model.addAttribute("posts", postService.findAllPosts());

return "admin";

}

@PostMapping("/users/delete/{id}")

public String deleteUser(@PathVariable Long id) {

userService.deleteUser(id);

return "redirect:/admin/dashboard";

}

}

**Deliverables**:

* Functional dashboard with post creation/viewing.
* Editable user profiles.
* Admin dashboard with user/post management.

**Phase 4: Contact Form & Messaging**

**Duration**: 1 week  
**Goals**: Implement support communication via email.

1. **Contact Page**
   * Create contact.html:
     + Form fields: Subject, message, user email (pre-filled for logged-in users).
   * Implement ContactController:

java

Copy

@Controller

public class ContactController {

@Autowired

private EmailService emailService;

@GetMapping("/contact")

public String showContactForm(Model model) {

model.addAttribute("message", new Message());

return "contact";

}

@PostMapping("/contact")

public String sendMessage(@ModelAttribute Message message, Authentication auth) {

emailService.sendContactEmail(message, auth.getName());

return "redirect:/contact?success";

}

}

1. **Email Integration**
   * Configure SendGrid in application.properties:

properties

Copy

spring.mail.host=smtp.sendgrid.net

spring.mail.port=587

spring.mail.username=apikey

spring.mail.password=${SENDGRID\_API\_KEY}

spring.mail.properties.mail.smtp.auth=true

spring.mail.properties.mail.smtp.starttls.enable=true

* + Create EmailService:

java

Copy

@Service

public class EmailService {

@Autowired

private JavaMailSender mailSender;

public void sendContactEmail(Message message, String username) {

SimpleMailMessage mail = new SimpleMailMessage();

mail.setTo("support@yourdomain.com");

mail.setSubject(message.getSubject());

mail.setText("From: " + username + "\n" + message.getContent());

mailSender.send(mail);

}

}

**Deliverables**:

* Working contact form.
* Emails sent via SendGrid to a support address.

**Phase 5: Payment Integration**

**Duration**: 1.5 weeks  
**Goals**: Enable Pro tier upgrades via PayPal.

1. **PayPal Setup**
   * Create a PayPal Sandbox account at [developer.paypal.com](https://developer.paypal.com/).
   * Add PayPal dependency to pom.xml:

xml

Copy

<dependency>

<groupId>com.paypal.sdk</groupId>

<artifactId>rest-api-sdk</artifactId>

<version>1.14.0</version>

</dependency>

* + Configure PayPal in application.properties:

properties

Copy

paypal.mode=sandbox

paypal.client.id=${PAYPAL\_CLIENT\_ID}

paypal.client.secret=${PAYPAL\_CLIENT\_SECRET}

1. **Payment Flow**
   * Create payment.html:
     + Button labeled "Upgrade to Pro" linking to PayPal.
   * Implement PaymentController:

java

Copy

@Controller

@RequestMapping("/payment")

public class PaymentController {

@Autowired

private PayPalService payPalService;

@GetMapping("/upgrade")

public String showPaymentPage() {

return "payment";

}

@PostMapping("/create")

public String createPayment() throws PayPalRESTException {

Payment payment = payPalService.createPayment(

10.00, "USD", "Pro Subscription",

"http://localhost:8080/payment/success",

"http://localhost:8080/payment/cancel"

);

return "redirect:" + payment.getLinks().stream()

.filter(link -> link.getRel().equals("approval\_url"))

.findFirst().get().getHref();

}

@GetMapping("/success")

public String paymentSuccess(@RequestParam("paymentId") String paymentId, @RequestParam("PayerID") String payerId, Authentication auth) {

payPalService.executePayment(paymentId, payerId);

userService.upgradeToPro(auth.getName());

return "redirect:/user/profile?pro=success";

}

}

1. **Pro Version Logic**
   * Update User entity with isPro field.
   * Restrict premium features (e.g., post visibility, analytics) to isPro == true.

**Deliverables**:

* PayPal payment flow working in Sandbox mode.
* Users upgraded to Pro status post-payment.

**Phase 6: Security & UX Enhancements**

**Duration**: 1 week  
**Goals**: Improve security and user experience.

1. **Password Visibility Toggle**
   * Add to login.html and register.html:

html

Copy

<input type="password" id="password" name="password">

<button type="button" onclick="togglePassword()">Show/Hide</button>

<script>

function togglePassword() {

const input = document.getElementById("password")*;*

input.type = input.type === "password" ? "text" : "password"*;*

}

</script>

1. **Security Enhancements**
   * Enable HTTPS in SecurityConfig:

java

Copy

http.requiresChannel().anyRequest().requiresSecure();

* + Configure CSRF protection (default in Spring Security).
  + Add rate limiting for login attempts (optional, using Bucket4j).

1. **Input Validation** *(Added Content)*:
   * Use Hibernate Validator for forms:

java

Copy

public class User {

@NotBlank

@Size(min = 3, max = 50)

private String username;

@Email

private String email;

@NotBlank

@Size(min = 8)

private String password;

}

* + Display validation errors in Thymeleaf templates.

**Deliverables**:

* Password toggle on login/registration pages.
* Secure production configuration.
* Validated user inputs.

**Phase 7: Testing & Documentation**

**Duration**: 1 week  
**Goals**: Ensure quality and document the project.

1. **Testing Strategy** *(Added Content)*:
   * **Unit Tests**:
     + Test services (UserService, PostService) using JUnit/Mockito.
     + Example:

java

Copy

@Test

public void testRegisterUser() {

User user = new User();

user.setUsername("test");

user.setPassword("password");

when(userRepository.save(any(User.class))).thenReturn(user);

userService.registerUser(user);

verify(userRepository).save(user);

}

* + **Integration Tests**:
    - Test API endpoints with @SpringBootTest.
  + **Manual Testing**:
    - Test registration, post creation, payment flow, and admin actions.
  + **Postman**:
    - Create a collection with endpoints:
      * POST /api/auth/register
      * POST /api/posts
      * GET /api/users

1. **Documentation**:
   * Update README.md:
     + Project overview, setup instructions, API endpoints.
     + Example:

markdown

Copy

## Social Media Platform

A Spring Boot app with user authentication, posts, and PayPal integration.

### Setup

1. Clone: `git clone ...`

2. Run: `mvn spring-boot:run`

* + Generate API docs using Springdoc OpenAPI (optional):

xml

Copy

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

<version>2.2.0</version>

</dependency>

**Deliverables**:

* Test coverage for critical features.
* Comprehensive README and Postman collection.

**Phase 8: Deployment**

**Duration**: 1 week  
**Goals**: Launch the platform publicly.

1. **Free Hosting Options**
   * **Heroku**:
     + Create Procfile:

Procfile

Copy

web: java -jar target/\*.jar --server.port=$PORT

* + - Deploy:

bash

Copy

heroku create your-app-name

heroku addons:create heroku-postgresql:hobby-dev

git push heroku main

* + - Set environment variables:

bash

Copy

heroku config:set SPRING\_PROFILES\_ACTIVE=prod

heroku config:set PAYPAL\_CLIENT\_ID=xxx

* + **Render**:
    - Connect GitHub repo.
    - Configure PostgreSQL add-on and environment variables.

1. **Production Database**
   * Configure application-prod.properties:

properties

Copy

spring.datasource.url=${JDBC\_DATABASE\_URL}

spring.datasource.username=${JDBC\_DATABASE\_USERNAME}

spring.datasource.password=${JDBC\_DATABASE\_PASSWORD}

spring.jpa.hibernate.ddl-auto=update

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

1. **Scalability Considerations** *(Added Content)*:
   * Use Heroku dynos to handle increased traffic.
   * Cache static assets (CSS/JS) with Spring Resource Handling.
   * Monitor performance with Sentry (free tier).

**Deliverables**:

* Live app (e.g., https://your-app.herokuapp.com).
* Production database connected.
* Environment variables secured.

**Tools & Free Services**

* **Version Control**: GitHub (free public repo).
* **CI/CD**: GitHub Actions (free tier, e.g., auto-build on push).

yaml

Copy

name: Build

on: [push]

jobs:

build:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Set up JDK 17

uses: actions/setup-java@v3

with: { java-version: '17' }

- name: Build with Maven

run: mvn clean install

* **Email**: SendGrid (100 emails/day free).
* **Monitoring**: Sentry (error tracking, free tier).
* **Testing**: Postman (free plan for API collections).

**Documentation & Live Demo**

* **GitHub Repository**: [Placeholder: Link to repo]. Include README with setup, screenshots, and contribution guidelines.
* **Postman Collection**: Export and share JSON collection for API testing.
* **Live Demo**: Deployed URL (e.g., https://your-app.herokuapp.com).
* **System Architecture Diagram** *(Added Content)*:
  + Create using [draw.io](https://draw.io):
    - Components: Web Browser (Thymeleaf), Spring Boot Backend, PostgreSQL, PayPal API, SendGrid API.
    - Flow: HTTP requests from browser to backend, JPA to database, HTTPS to external APIs.
    - Export as PNG and include in README/doc.

**Next Steps**

1. Start Phase 1 today: Initialize project and push to GitHub.
2. Follow incremental development:
   * Week 1: Setup and authentication.
   * Week 2-3: Profile and posts.
   * Week 4: Contact form.
   * Week 5: Payments.
   * Week 6: Security/UX.
   * Week 7: Testing/docs.
   * Week 8: Deployment.
3. Test locally after each feature (use mvn spring-boot:run).
4. Commit changes regularly with clear messages (e.g., git commit -m "Add user registration").

**Changes & Additions Made**

1. **Timelines**: Added duration estimates for each phase (8-10 weeks total).
2. **Testing Strategy**: Included unit, integration, and manual testing details with code examples.
3. **Admin Features**: Added admin dashboard and user/post management in Phase 3.
4. **Scalability**: Noted Heroku dynos, caching, and monitoring in Phase 8.
5. **Validation**: Added Hibernate Validator for form inputs in Phase 6.
6. **CI/CD**: Included GitHub Actions workflow for automated builds.
7. **Architecture Diagram**: Referenced diagram creation in Documentation section, linking to prior discussion.
8. **Code Snippets**: Expanded examples (e.g., PaymentController, EmailService) for clarity.
9. **Environment Profiles**: Added application-prod.properties for production.
10. **Error Handling**: Implied error pages (error.html) and validation feedback.

**How to Use with DOCX**

Since you previously requested a DOCX, you can integrate this roadmap into your documentation:

1. Copy the markdown above.
2. Paste into Microsoft Word or Google Docs.
3. Format headings (# as Heading 1, ## as Heading 2).
4. Convert code blocks to monospaced font (e.g., Consolas) with gray background.
5. Insert the system architecture diagram (from prior response) as a PNG.
6. Save as .docx.

**If you need the diagram**:

* Use the detailed description I provided earlier (Web Browser → Spring Boot → PostgreSQL → PayPal/SendGrid).
* Create in [draw.io](https://draw.io) and export as PNG.
* I can provide further draw.io steps if needed.

**If you need specific snippets**:

* Let me know (e.g., full PayPalService, Thymeleaf templates, or test cases).

## ✅ **Current Good Points:**

* **Technology Stack:** Clearly defined (Spring Boot, PostgreSQL, Thymeleaf, Spring Security, SendGrid, PayPal).
* **Database Design:** Well-defined schemas and ERD.
* **Deployment:** Steps for Heroku clearly explained (but Render is mentioned without specifics).
* **Security:** Uses BCrypt, HTTPS enforcement, and CSRF protection.
* **Payments:** Clearly defined PayPal integration.
* **Testing:** Strategy includes JUnit, Mockito, and Postman.
* **CI/CD:** Mentioned GitHub Actions for automation.

## ⚠️ **Gaps & Recommendations for Render Deployment:**

### **Step-by-step setup for Render deployment**

1. **Create a GitHub Repo**:
   * Push your code to GitHub.
2. **Render Account Setup**:
   * Sign up at [render.com](https://render.com).
3. **Create Web Service in Render**:
   * Click "New" → "Web Service".
   * Connect your GitHub repository.
4. **Render Configuration**:
   * **Environment:** Select "Java".
   * **Build Command:** Ensure the following build command:

bash

CopyEdit

./mvnw clean install -DskipTests

(Note: Include Maven wrapper (*mvnw*) in your repo for consistent builds.)

* + **Start Command**:

bash

CopyEdit

java -jar target/\*.jar

* + **Environment Variables**: Add these essential environment variables securely:
    - PAYPAL\_CLIENT\_ID
    - PAYPAL\_CLIENT\_SECRET
    - SENDGRID\_API\_KEY
    - JDBC\_DATABASE\_URL
    - JDBC\_DATABASE\_USERNAME
    - JDBC\_DATABASE\_PASSWORD
    - SPRING\_PROFILES\_ACTIVE=prod

1. **Database Setup in Render**:
   * Create a PostgreSQL database directly via Render’s add-on or connect to external PostgreSQL.
   * Render automatically generates environment variables (DATABASE\_URL) for PostgreSQL. Map these variables appropriately:

properties

CopyEdit

spring.datasource.url=${DATABASE\_URL}

1. **Health Checks and Zero Downtime Deploys**:
   * Setup Render health checks (GET /actuator/health).
   * Ensure you include Spring Boot Actuator dependency in your pom.xml:

xml

CopyEdit

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

* + Configure application-prod.properties:

properties

CopyEdit

management.endpoints.web.exposure.include=health,info

management.endpoint.health.show-details=always

## 🚩 **Additional Recommendations:**

### ✅ **Improvement of Architecture & Scalability:**

* **Caching:**
  + Implement Redis for caching frequently accessed posts/user profiles for performance optimization.
  + Include Redis support with:

xml

CopyEdit

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

* **Rate Limiting:**
  + Add Bucket4j to avoid brute force login attempts or API abuse.

### ✅ **Real-time Features:**

* Consider adding real-time capabilities (notifications, messaging) with WebSocket/STOMP.

### ✅ **File Uploads:**

* Integrate AWS S3 for scalable and secure file storage (for images, user-generated content).

xml

CopyEdit

<dependency>

<groupId>software.amazon.awssdk</groupId>

<artifactId>s3</artifactId>

<version>2.20.43</version>

</dependency>

## 🚩 **Quality and Security Assurance:**

* **Logging and Monitoring:**
  + Use Sentry or New Relic for error tracking and performance monitoring.
  + Configure structured logging (Logback/SLF4J).
* **Security Enhancements:**
  + Enable security headers (Content-Security-Policy, X-Frame-Options, X-XSS-Protection).
  + Verify HTTPS and SSL enforcement in Render.

## ✅ **CI/CD Enhancements:**

* Enhance GitHub Actions workflow to include automated deployments:

yaml

CopyEdit

name: Deploy to Render

on:

push:

branches: [ main ]

jobs:

build:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v4

- uses: actions/setup-java@v4

with:

java-version: '17'

distribution: 'temurin'

- run: ./mvnw clean install

deploy:

needs: build

runs-on: ubuntu-latest

steps:

- name: Trigger Render Deployment

run: |

curl -X POST "${{ secrets.RENDER\_DEPLOY\_HOOK }}"

(Note: Setup deploy hook in Render and store as GitHub secret.)

## 🚩 **Comprehensive Documentation & Diagrams:**

* Replace placeholder diagrams in your documentation:
  + System architecture
  + Entity Relationship Diagram (ERD)
* Clearly document configuration steps for local vs. production environments.

## ⚙️ **Recommended Directory Structure Enhancements:**

For clarity, consider slightly enhancing the directory structure to clearly separate concerns:

bash

CopyEdit

src/main/

├── java/com/yourdomain/

│ ├── config/

│ ├── controller/

│ ├── model/

│ ├── repository/

│ ├── service/

│ ├── util/ # Add utilities like helpers, validators, etc.

│ └── Application.java

└── resources/

├── static/

├── templates/

├── application.properties

├── application-prod.properties

└── db/migration/ # Use Flyway or Liquibase for DB migrations

## 📌 **Summary of Missing Elements & Recommendations:**

* **Detailed Render deployment instructions**.
* **Caching (Redis)** for performance.
* **AWS S3** for file uploads.
* **Enhanced security headers**.
* **Advanced logging & monitoring** tools.
* **Real-time messaging features** (optional enhancement).
* **Comprehensive CI/CD pipeline** integrated with Render deploy hooks.
* **Replace placeholders with detailed diagrams and illustrations**.