Online Quiz Platform Project Task Specification

Spring Boot REST API Development Task

Assigned by: Project Leader

Date: September 11, 2025

Version 1.0

Contents

1 Project Overview

The Online Quiz Platform is a RESTful web application built with Spring Boot, designed to enable users to create, take, and manage quizzes. Instructors can design quizzes with multiple-choice questions, students can attempt quizzes and view scores, and administrators can moderate content. The system incorporates real-time updates, leaderboards, and robust security to provide a scalable and secure platform.

2 Objectives

- Develop a scalable REST API to manage quizzes, questions, user accounts, and scores.
- Utilize advanced Spring features, including Spring Security, Spring Data JPA, Spring WebSocket, and Spring Cache.
- Implement real-time updates for quiz results and leaderboards.
- Provide comprehensive API documentation and testing to ensure reliability and maintainability.

3 Functional Requirements

3.1 User Roles

- **Student**: Register, log in, browse quizzes, take quizzes, view scores, and check leaderboards.
- **Instructor**: Create, update, and delete quizzes and questions; view student performance.
- **Admin**: Approve quizzes, manage users, and monitor system health.

3.2 Core Features

1. **User Management**:
    - Register and authenticate users with email and password.
    - Support role-based access (Student, Instructor, Admin).
2. **Quiz Management**:
    - Create, update, delete, and list quizzes (title, description, category).
    - Add multiple-choice questions with correct/incorrect options.
3. **Quiz Taking**:
    - Allow students to start a quiz, submit answers, and receive immediate scores.
    - Implement time-limited quizzes (e.g., 30 minutes per quiz).
4. **Leaderboard**:
    - Display top performers globally or per quiz based on scores.
5. **Real-Time Updates**:
    - Notify users of quiz results or leaderboard changes in real time.

6. **Admin Dashboard**:
    - Approve/reject quizzes created by instructors.
    - View system metrics (e.g., active users, quiz completion rates).

4 Technical Requirements

4.1 Technology Stack

- **Framework**: Spring Boot 3.x
- **Database**: PostgreSQL (or MySQL) for relational data; Redis for caching.
- **Authentication**: OAuth2 or JWT for securing APIs.
- **Real-Time**: Spring WebSocket with STOMP for real-time updates.
- **Caching**: Spring Cache with Redis for leaderboard and quiz data.
- **API Documentation**: Spring REST Docs or Swagger/OpenAPI.
- **Testing**: JUnit, Mockito, and TestRestTemplate for unit and integration tests.

4.2 Spring Features

- **Spring Boot Web**: REST APIs with `@RestController` for CRUD operations.
- **Spring Data JPA**: Entity management for `User`, `Quiz`, `Question`, `Answer`, and `Score`.
- **Spring Security**: Role-based authentication and authorization.
- **Spring WebSocket**: Real-time notifications for quiz results and leaderboards.
- **Spring Cache**: Cache frequently accessed data (e.g., quiz questions, leaderboards).
- **Spring Boot Actuator**: Monitor application health and metrics.
- **Spring REST Docs**: Generate API documentation during testing.

4.3 Database Schema

- **User**: id, email, password, role (STUDENT/INSTRUCTOR/ADMIN), name.
- **Quiz**: id, title, description, category, createdBy (Instructor), approved (boolean).
- **Question**: id, quizId, text, options (JSON or separate table), correctOption.
- **Answer**: id, userId, quizId, questionId, selectedOption, timestamp.
- **Score**: id, userId, quizId, score, completionTime.
- **Leaderboard**: id, quizId, userId, score, rank (optional, can be derived).

4.4 API Endpoints

- **User Management**:
    - `POST /api/auth/register` – Register a new user.
    - `POST /api/auth/login` – Authenticate and return JWT.
    - `GET /api/users/{id}` – Get user details (admin/instructor only).

- **Quiz Management**:
  - `POST /api/quizzes` – Create a quiz (instructor only).
  - `GET /api/quizzes` – List approved quizzes (paginated).
  - `PUT /api/quizzes/{id}` – Update quiz details (instructor only).
  - `DELETE /api/quizzes/{id}` – Delete a quiz (instructor/admin).
- **Question Management**:
  - `POST /api/quizzes/{id}/questions` – Add a question to a quiz.
  - `GET /api/quizzes/{id}/questions` – List questions for a quiz.
- **Quiz Taking**:
  - `POST /api/quizzes/{id}/start` – Start a quiz session.
  - `POST /api/quizzes/{id}/submit` – Submit answers and get score.
- **Leaderboard**:
  - `GET /api/leaderboard/global` – Get global leaderboard.
  - `GET /api/leaderboard/quiz/{id}` – Get quiz-specific leaderboard.
- **Admin**:
  - `PUT /api/quizzes/{id}/approve` – Approve/reject a quiz.
  - `GET /api/admin/metrics` – View system metrics (via Actuator).

5 Non-Functional Requirements

- **Scalability**: Use Redis caching for leaderboards and quiz data to handle high traffic.
- **Security**: Secure all endpoints with JWT/OAuth2; validate inputs with Spring Validation.
- **Performance**: Ensure API response time < 200ms for cached data; use pagination for large datasets.
- **Reliability**: Implement retry mechanisms for external integrations (if any).
- **Documentation**: Provide comprehensive API documentation using Spring REST Docs or Swagger.

6 Deliverables

1. **Source Code**:
   - Spring Boot project hosted on GitHub with a clear README.
   - Organized into packages (e.g., `controller`, `service`, `repository`, `model`).
2. **API Documentation**:
   - Generated using Spring REST Docs or Swagger, covering endpoints, request/response formats, and error codes.
3. **Tests**:

- Unit tests for service layer (80%+ coverage using JUnit/Mockito).
- Integration tests for REST endpoints using TestRestTemplate.

4. **Database Setup**:
   - SQL scripts for initializing PostgreSQL schema.
   - Configuration for Redis caching.
5. **Deployment**:
   - Dockerized application with a `docker-compose.yml` for local testing.
   - Instructions for deploying to a cloud provider (e.g., AWS, Heroku).
6. **Demo**:
   - Postman collection or video demo showcasing key features (e.g., quiz creation, taking a quiz, leaderboard).

7 Optional Enhancements

- **Microservices**: Split into microservices (e.g., User Service, Quiz Service) using Spring Cloud, Eureka, and Spring Cloud Gateway.
- **Real-Time Quiz**: Implement live quizzes with multiple users using Spring WebSocket.
- **Search Functionality**: Add quiz search by title/category using Spring Data JPA Specifications or Elasticsearch.
- **Notifications**: Send email/SMS notifications for quiz results using Spring Integration (e.g., SendGrid).
- **Analytics**: Generate quiz performance analytics (e.g., average score per quiz) using Spring Data JPA.
- **Mobile App Integration**: Expose APIs for a mobile app and test with a simple frontend (e.g., React Native).

8 Milestones and Timeline

Total Duration: 4–6 weeks

- **Week 1: Setup and User Management**
  - Set up Spring Boot project with dependencies.
  - Implement user registration, login, and role-based authentication.
  - Design database schema and initialize PostgreSQL.
  - *Deliverable*: Working user management APIs with JWT authentication.
- **Week 2: Quiz and Question Management**
  - Implement CRUD APIs for quizzes and questions.
  - Add validation and role-based access.
  - Set up Redis caching for quiz lists.

  - *Deliverable*: Quiz and question APIs with caching.
- **Week 3: Quiz Taking and Scoring**
  - Implement quiz-taking logic with time limits and scoring.
  - Create APIs for submitting answers and retrieving scores.
  - Add WebSocket support for real-time score updates.
  - *Deliverable*: Functional quiz-taking feature with real-time updates.
- **Week 4: Leaderboard and Admin Features**
  - Implement leaderboard APIs with caching.
  - Add admin APIs for quiz approval and system metrics.
  - Set up Spring Boot Actuator for monitoring.
  - *Deliverable*: Leaderboard and admin dashboard APIs.
- **Week 5–6 (Optional)**:
  - Implement stretch goals (e.g., microservices, search, notifications).
  - Write comprehensive tests and documentation.
  - Dockerize and deploy to a cloud provider.
  - *Deliverable*: Final project with documentation, tests, and deployment.

9 Success Criteria

- All core features (user management, quiz management, quiz taking, leaderboard, admin dashboard) are fully functional.
- APIs are secure, performant, and well-documented.
- At least 80% test coverage for critical components.
- Application runs in Docker with no errors and supports 100+ concurrent users.
- Code follows clean architecture principles (e.g., separation of concerns, REST best practices).

10 Resources and Guidelines

- **Spring Initializr**: Use `https://start.spring.io` to set up the project.
- **Best Practices**: Follow REST API design principles (appropriate HTTP methods, status codes).
- **Inspiration**: Explore GitHub repositories for Spring Boot quiz apps (search “spring boot quiz application”).
- **Tools**:
  - **Database**: PostgreSQL for persistence, Redis for caching.
  - **Testing**: Postman for API testing, JUnit/Mockito for unit tests.
  - **Monitoring**: Spring Boot Actuator with Prometheus/Grafana (optional).

- **Documentation**: Use Spring REST Docs or Swagger for professional API docs.

11 Sample Code

11.1 Quiz Controller

```
@RestController
@RequestMapping("/api/quizzes")
public class QuizController {
    private final QuizService quizService;

    @Autowired
    public QuizController(QuizService quizService) {
        this.quizService = quizService;
    }

    @PostMapping
    @PreAuthorize("hasRole('INSTRUCTOR')")
    public ResponseEntity<QuizDTO> createQuiz(@Valid @RequestBody QuizDTO
        quizDTO) {
        QuizDTO createdQuiz = quizService.createQuiz(quizDTO);
        return ResponseEntity.status(HttpStatus.CREATED).body(createdQuiz);
    }

    @GetMapping
    public ResponseEntity<Page<QuizDTO>> getQuizzes(Pageable pageable) {
        return ResponseEntity.ok(quizService.getApprovedQuizzes(pageable));
    }
}
```

11.2 Quiz Entity

```
@Entity
public class Quiz {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private String description;
    private String category;
    @ManyToOne
    private User createdBy;
    private boolean approved;

    @OneToMany(mappedBy = "quiz")
    private List<Question> questions;
    // Getters and setters
}
```

11.3 Security Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
        throws Exception {
```

```
        http
            .csrf().disable()
            .authorizeHttpRequests()
            .requestMatchers("/api/auth/**").permitAll()
            .requestMatchers("/api/quizzes/**").hasAnyRole("INSTRUCTOR", "
                ADMIN")
            .anyRequest().authenticated()
            .and()
            .sessionManagement().sessionCreationPolicy(
                SessionCreationPolicy.STATELESS)
            .and()
            .addFilterBefore(jwtAuthenticationFilter(),
                UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }
}
```

12 Support and Clarifications

- **Weekly Check-Ins**: Submit progress updates (e.g., GitHub commits, API demos) weekly for feedback.
- **Questions**: Reach out for blockers (e.g., WebSocket setup, Redis integration) for code snippets or guidance.
- **Code Reviews**: Share code for critical components (e.g., security, WebSocket) for review.

13 Next Steps

1. **Setup**: Initialize the project using Spring Initializr with dependencies (Web, Data JPA, Security, WebSocket, Redis, Actuator).
2. **Database**: Create the schema in PostgreSQL and configure `application.yml`.
3. **Start Coding**: Begin with user management and authentication (Week 1 goals).
4. **Questions**: Request clarification for specific features (e.g., WebSocket, Redis) as needed.