***Shiv Sunder Dash Report***

Secure, Maintainable & Tested REST API using Spring Boot

Assignment 1: Implement Coding Best Practices in a Books/Library REST API Service

I have implemented a Spring Boot REST API for managing a library’s book inventory, following coding best practices throughout the project. The API uses clear Java naming conventions for classes, methods, and variables to maintain readability. All classes are kept small and single-purpose, with service and repository layers separating concerns. Exception handling is centralized using a @ControllerAdvice class to return structured error responses. Method-level comments and JavaDocs are provided to explain business logic and usage.

The project also follows Spring Boot and microservices best practices. The design uses DTO classes to decouple entity persistence from API contracts, with validation annotations to enforce constraints. Properties are externalized for different profiles, and REST endpoints follow proper resource naming. Design patterns such as Service and Repository patterns structure the app, while DTO and Factory patterns simplify data transfer and test data creation. This ensures maintainable, scalable code that meets professional standards.

Assignment 2: Implement OWASP Top 10 and Security Concepts

I have implemented security measures in the Books/Library REST API to address OWASP Top 10 vulnerabilities and demonstrate secure coding concepts. SQL Injection risks are mitigated by using Spring Data JPA repositories with parameterized queries. XSS protections are implemented through input validation and output encoding. Broken authentication is addressed by using Spring Security’s login flow with BCrypt password hashing, ensuring passwords are not stored in plaintext.

CORS configuration is customized to allow only trusted origins and methods. For token-based authentication, I implemented JWTs to secure REST API calls, adding a filter to validate tokens on protected endpoints. HTTPS enforcement is simulated by redirecting HTTP to HTTPS in production profiles. API rate limiting is demonstrated via a custom filter that restricts repeated requests from the same client. The login scenario securely encodes passwords with BCryptPasswordEncoder and verifies hashes on login to prevent credential theft and reuse.

Assignment 3: Implement Testing and Code Quality Best Practices

I have implemented automated testing and code quality checks to maintain a robust codebase for the REST API service. Unit and integration testing use JUnit and Spring Boot’s testing features. Controllers are tested with @WebMvcTest using MockMvc to verify responses and error codes. Full application context tests use @SpringBootTest to ensure that beans and configuration work as expected. Repository integration tests use TestContainers to spin up real database instances for isolated, production-like testing.

For code coverage, the JaCoCo plugin is configured to generate HTML reports and enforce coverage thresholds. Static code analysis uses PMD, FindBugs (SpotBugs), and Checkstyle plugins to catch style violations and potential bugs early. SonarQube for IDE is installed in IntelliJ to provide live feedback on code smells and vulnerabilities, guiding fixes during development. This setup ensures maintainable, secure, and well-tested code, following industry-standard best practices for professional Java development.

- SonarQube for IDE previously known as sonarlint:  
 Used IntelliJ plugin to fix warnings in real time.

Main repository:

<https://github.com/Shiv-Sunder-Dash/Sprint5_Day2>

Git clone:

<https://github.com/Shiv-Sunder-Dash/Sprint5_Day2.git>

1. C. Adopt Optional and fail-fast validations
2. B. Programming to interfaces, not implementations
3. C. It makes classes harder to test and refactor
4. B. Organize by layer (controller, service, repository, config, etc.)
5. D. It may delete parent records unexpectedly
6. B. Use versioning (e.g., /api/v1)
7. C. Use a discovery service like Eureka, Consul, or Kubernetes DNS
8. D. To ensure autonomy and prevent tight coupling
9. D. Use Retry + Timeout + Circuit Breaker concepts
10. C. Use centralized logging with correlation IDs (e.g., ELK, Loki)
11. B. Creating objects in loops unnecessarily
12. C. Open/Closed Principle
13. D. Difficult unit testing and immutability issues
14. C. Aggregate responses, route requests, and handle cross-cutting concerns
15. B. Optimize for graceful shutdown and fast startup
16. Returning domain entities directly in API response
17. Add security config and redirect HTTP to HTTPS using HttpSecurity
18. Content-Security-Policy
19. YAML or JSON
20. C. For controller layer testing with mocked services