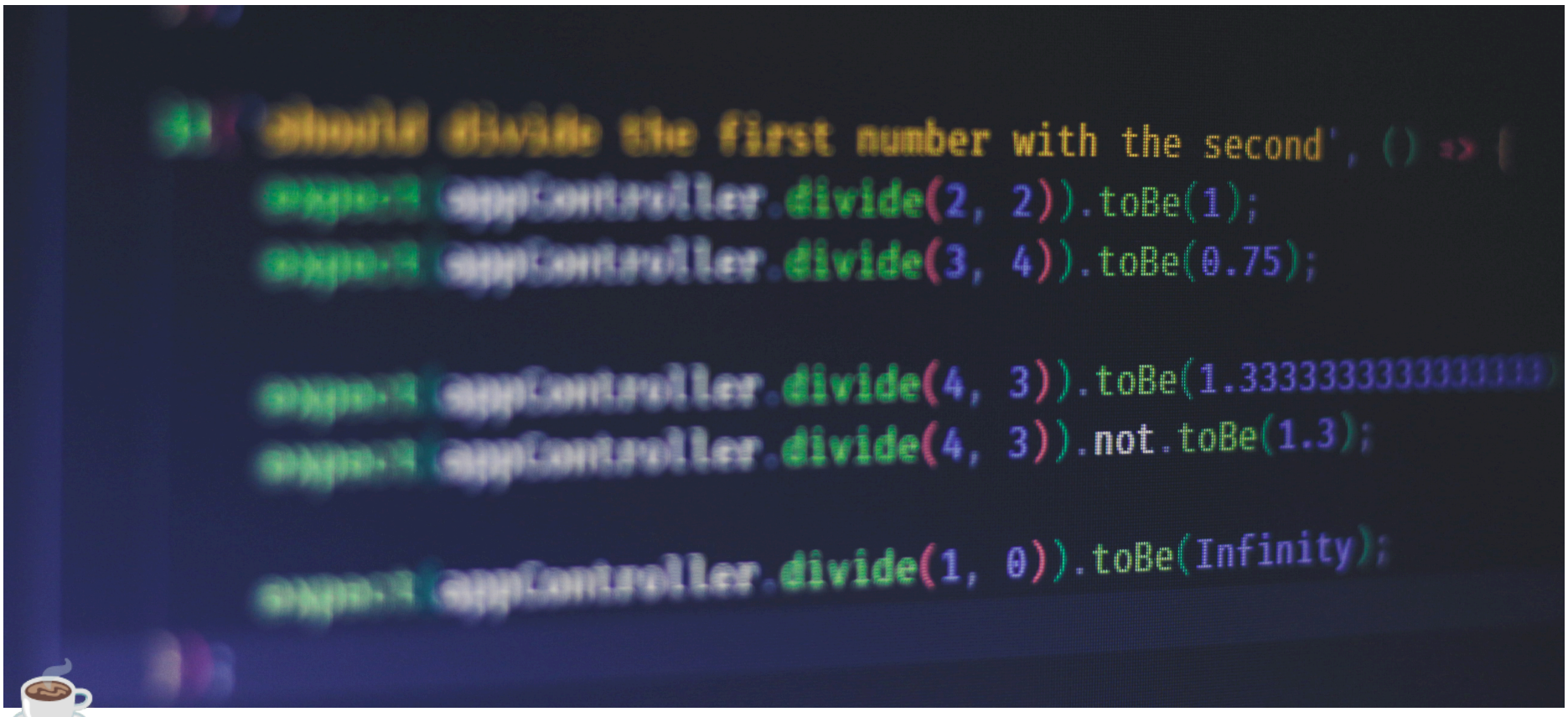


Mohammadreza Motallebi - ATTSW Project Report - 2025

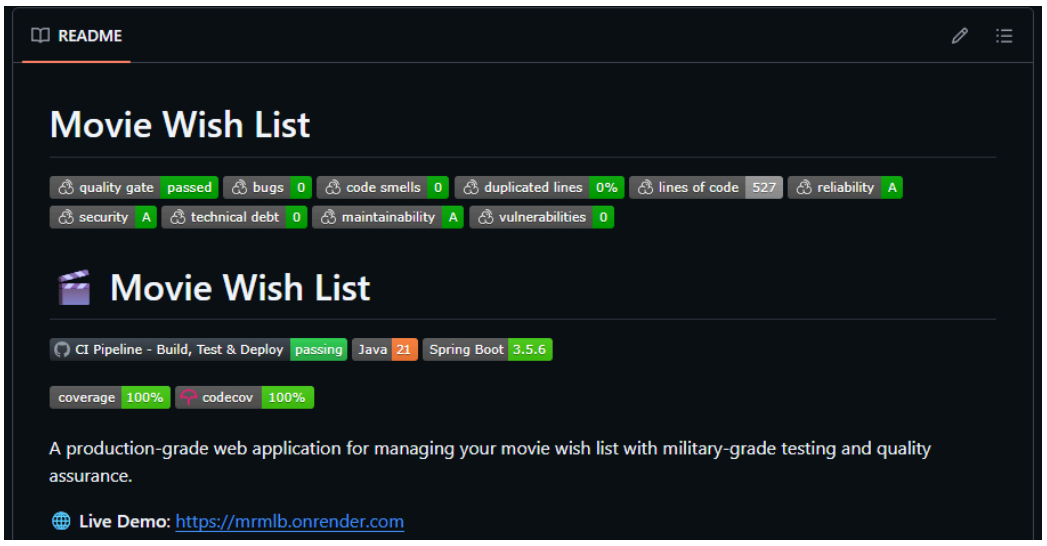
Movie Wish List | Aligned with the course materials

ADVANCED TECHNIQUES AND TOOLS FOR SOFTWARE DEVELOPMENT | B027540(B059)

1. Introduction

This technical report documents the implementation of the **Movie Wish List project**, a production-grade Spring Boot web application developed following Test-Driven Development (TDD) principles and industry best practices. The project demonstrates comprehensive testing strategies, Maven build automation, continuous integration, and containerized deployment.

As stated in the main textbook , modern software development requires a systematic approach to testing that follows the **test pyramid structure**: unit tests at the base, integration tests in the middle, and end-to-end tests at the top. This project strictly adheres to this architecture, ensuring high code quality through automated testing and quality gates.



1.1 Project Overview

The Movie Wish List application is a full-stack web application built with **Spring Boot 3.5.6** and **Java 21**, allowing users to manage a list of movies they wish to watch. The application features:

- RESTful API endpoints for CRUD operations on movie entities
- Persistent storage using **MongoDB Atlas** (cloud database)
- Responsive web interface with Thymeleaf templates
- Docker containerization for consistent deployment
- Comprehensive test coverage with **100% code coverage** and **100% mutation coverage** requirements

1.2 Technology Stack

KISS Principle: Keep It Simple, Stupid

According to the exam guidelines and the TDD textbook (page 76), this project strictly follows the **KISS principle**. As explicitly stated in the course instructions:

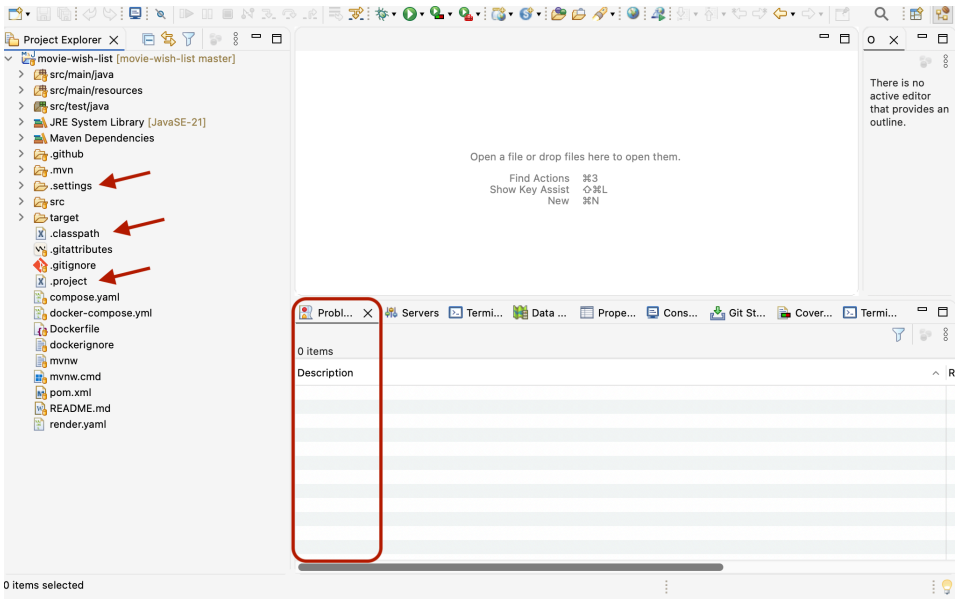
> "Keep the application and its code simple and concentrate on all the 'surrounding' mechanisms and techniques."

The project deliberately implements a **minimalist domain model** with only two classes (`Wishlist` and `WishlistRepository`) to focus on demonstrating test-driven development, build automation, and continuous integration practices rather than business logic complexity . This approach aligns with the guideline's recommendation:

> "The complexity of the main code is not essential (indeed, we suggest you implement a simple application)."

The guideline suggests domain models like "Todo with description and boolean field" or "Student and Teacher" – this project implements a **Movie Wish List** with similar simplicity: title, description, tags, and a done flag . This allows comprehensive coverage of testing strategies (unit, integration, E2E), code quality tools (JaCoCo, PIT), and CI/CD pipelines without the overhead of complex business rules .

Warning and Errors - 0!



Spring Boot Framework Selection

According to the Spring Boot First Project tutorial, Spring Boot provides powerful auto-configuration mechanisms that simplify application setup .

The project leverages this capability through the following stack :

Backend Framework:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::                (v3.5.6)
```

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.5.6</version>
</parent>
```

As explained in the course materials (Spring Boot First Project), using the Spring Boot parent POM provides curated dependency versions and pre-configured Maven plugins . This ensures all libraries work together harmoniously, eliminating dependency conflicts .

Core Dependencies (KISS in Action):

The project uses only **essential dependencies**, avoiding unnecessary complexity :

- `spring-boot-starter-web`: Provides Spring MVC, embedded Tomcat, and JSON support
- `spring-boot-starter-data-mongodb`: MongoDB integration with Spring Data repositories
- `spring-boot-starter-thymeleaf`: Server-side template engine for webpages
- `spring-boot-starter-actuator`: Production-ready health monitoring endpoints

Testing Dependencies (Test Pyramid Focus):

Following the test pyramid architecture (pages 1-4 of the textbook), testing dependencies outnumber production dependencies :

- `spring-boot-starter-test`: JUnit 5, Mockito, AssertJ, Spring Test
- `spring-boot-testcontainers`: Integration with Testcontainers
- **`rest-assured` (5.5.6)**: BDD-style REST API testing framework for E2E tests (REST Tutorial , Text Book)
- **`awaitility` (4.2.2)**: Asynchronous system testing with intelligent waiting strategies (Text Book: 457)
- `testcontainers:mongodb`: MongoDB container for integration tests
- `selenium-java`: WebDriver for E2E testing
- `webdrivermanager`: Automatic browser driver management

This demonstrates the course principle: **simple application, comprehensive testing**.

Java 21 LTS Selection

The project uses Java 21 (Long-Term Support) rather than Java 8, 11, or 17 . While the old exam guidelines specify Java 8, 11, or 17 as allowed versions, Java 21 (released September 2023) is now the current LTS version and provides:

- Virtual Threads (Project Loom) for improved concurrency
- Pattern Matching enhancements for cleaner code
- Record Patterns for data-oriented programming
- Sequenced Collections for predictable iteration order

Note: The GitHub Actions CI pipeline tests only Java 21 (not a build matrix) to maintain simplicity and focus on testing strategies rather than cross-version compatibility .

This aligns with KISS: **one Java version, thoroughly tested**.

MongoDB Over SQL Database

The project uses MongoDB (NoSQL) instead of a relational database to maintain simplicity :

- **No schema migrations**: Collections created automatically
- **No ORM complexity**: Spring Data MongoDB provides repositories with zero boilerplate
- **Docker-friendly**: `mongo:7` image starts instantly without configuration
- **Test-friendly**: Testcontainers MongoDB container boots in <2 seconds

This choice embodies KISS: **simple data model, simple persistence, focus on testing**.

---

2. Project Architecture

2.1 Application Structure

Following the Spring Boot architectural patterns described in the course materials (Spring Boot First Project , REST Tutorial), the application follows a layered architecture with clear separation of concerns :

```
com.example.movie.wish.list/
├── controller/          # REST controllers
├── service/             # Business logic layer
├── repository/          # Data access layer
├── model/               # Domain entities
└── MovieWishListApplication.java
```

2.2 Domain Model

The `Wishlist` entity is annotated with `@Document`, marking it as a MongoDB document, as described in the Spring Data MongoDB documentation :

```
@Document(collection = "moviesCollection")
public class Wishlist {
    @Id
    private String id;
    private String title;
    private String description;
    private List<String> tags;
    private boolean done;

    public Wishlist() {
        this.tags = new ArrayList<>();
    }

    public Wishlist(String title, String description, List<String> tags, boolean done) {
        this.title = title;
        this.description = description;
        this.tags = tags != null ? tags : new ArrayList<>();
        this.done = done;
    }

    // Getters and setters...
}
```

According to the Spring Boot tutorial (page 7), the `@Id` annotation specifies the primary key, which MongoDB generates automatically.

Note: The `Wishlist` entity is not exposed directly over REST; the API uses `WishlistDTO` , and the controller maps between DTO and entity.

2.3 Repository Layer

As taught in the course materials (Spring Boot First Project), Spring Data repositories can be declared as simple interfaces, and Spring creates implementations on-the-fly at runtime :

```
public interface WishlistRepository extends MongoRepository<Wishlist, String> { }
```

This interface extends `MongoRepository`, providing built-in CRUD methods like `findAll()`, `findById()`, `save()`, and `deleteById()` without requiring explicit implementations.

2.4 Service Layer

Following best practices from the Spring Boot Unit Tests tutorial, business logic is encapsulated in a service layer that depends on the repository through **constructor injection** :

```
@Service
public class WishlistService {
    private final WishlistRepository repository;

    public WishlistService(WishlistRepository repository) {
        this.repository = repository;
    }

    public Wishlist addMovie(Wishlist wishlist) {
        return repository.save(wishlist);
    }

    public List<Wishlist> getAllMovies() {
        return repository.findAll();
    }

    public Optional<Wishlist> getMovieById(String id) {
        return repository.findById(id);
    }

    public Wishlist updateMovie(String id, Wishlist updatedMovie) {
        return repository.findById(id)
                .map(existing -> {
                    existing.setTitle(updatedMovie.getTitle());
                    existing.setDescription(updatedMovie.getDescription());
                    existing.setTags(updatedMovie.getTags());
                    existing.setDone(updatedMovie.isDone());
                    return repository.save(existing);
                })
                .orElseThrow(() -> new RuntimeException("Movie not found"));
    }

    public void deleteMovie(String id) {
        repository.deleteById(id);
    }
}
```

As emphasized in the course materials (Spring Boot First Project), **constructor injection is preferred** over field injection ,

because it makes testing easier and dependencies explicit.

2.5 REST Controller

According to the REST Tutorial and Spring Boot materials , REST controllers handle HTTP requests and return JSON responses :

```
@RestController
@RequestMapping("/api/wishlist")
@CrossOrigin(origins = "*")
public class WishlistRestController {

    private final WishlistService service;

    public WishlistRestController(WishlistService service) {
        this.service = service;
    }

    @PostMapping
    public ResponseEntity<WishlistDTO> create(@RequestBody WishlistDTO dto) {
        Wishlist saved = service.addMovie(toEntity(dto));
```

```
    return ResponseEntity.ok(toDto(saved));
}

@GetMapping
public List<WishlistDTO> getAll() {
    return service.getAllMovies().stream()
            .map(this::toDto)
            .collect(Collectors.toList());
}

@GetMapping("/{id}")
public WishlistDTO getById(@PathVariable String id) {
    return service.getMovieById(id)
            .map(this::toDto)
            .orElseThrow(() -> new RuntimeException("Movie not found"));
}

@PutMapping("/{id}")
public WishlistDTO update(@PathVariable String id, @RequestBody WishlistDTO dto) {
    Wishlist updated = service.updateMovie(id, toEntity(dto));
    return toDto(updated);
}

@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void delete(@PathVariable String id) {
    service.deleteMovie(id);
}

// --- Mapping helpers (KISS) ---
private Wishlist toEntity(WishlistDTO dto) {
    Wishlist w = new Wishlist(
        dto.getTitle(),
        dto.getDescription(),
        dto.getTags(),
        dto.isDone()
    );
    return w;
}

private WishlistDTO toDto(Wishlist entity) {
    return new WishlistDTO(
        entity.getTitle(),
        entity.getDescription(),
        entity.getTags(),
        entity.isDone()
    );
  }
}
```

The `@RestController` annotation combines `@Controller` and `@ResponseBody`, automatically serializing return values to JSON (REST Tutorial)

2.6 DTO Layer & API Contracts

Rationale

At the API boundary we use Data Transfer Objects (DTOs) instead of exposing persistence entities. DTOs decouple the public REST contract from the internal domain model, reduce the risk of unintentionally leaking internal fields, and make it easier to evolve the API independently from database details. DTOs are also the natural place to apply input validation and to shape responses for clients. (KISS still applies: a single simple WishlistDTO is sufficient for this project.) Design. The DTO mirrors the fields the API accepts/returns (title, description, tags, done) and is kept free of persistence annotations. A no-args constructor supports Jackson deserialization; standard getters/setters support Jackson and testing.

```
package com.example.movie.wish.list.dto;

import java.util.List;
```

```
/**
 * Data Transfer Object for Wishlist API requests/responses.
 * Decouples external API from internal persistence entity.
 */
public class WishlistDTO {
    private String title;
    private String description;
    private List<String> tags; // e.g., ["sci-fi", "classic"]
    private boolean done;

    // Default constructor (required by Jackson)
    public WishlistDTO() {}

    // Convenience constructors
    public WishlistDTO(String title, String description) {
        this.title = title;
        this.description = description;
    }

    public WishlistDTO(
    String title,
    String description,
    List<String> tags,
    boolean done
    ) {
        this.title = title;
        this.description = description;
        this.tags = tags;
        this.done = done;
    }

    // Getters & setters
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    public String getDescription() { return description; }
    public void setDescription(String description)
    { this.description = description; }

    public List<String> getTags() { return tags; }
    public void setTags(List<String> tags) { this.tags = tags; }

    public boolean isDone() { return done; }
    public void setDone(boolean done) { this.done = done; }
}
```

Mapping

To keep things simple (KISS), mapping is implemented as private helper methods in the controller. For larger projects a dedicated mapper (or MapStruct) can be introduced, but manual mapping is sufficient here.

💡 ***REST Controller refactored to use DTOs***

WishlistRestController.java

```
@RestController
@RequestMapping("/api/wishlist")
@CrossOrigin(origins = "*")
public class WishlistRestController {

    private final WishlistService service;

    public WishlistRestController(WishlistService service) {
        this.service = service;
```

```
}

@PostMapping
public ResponseEntity<WishlistDTO> create(@RequestBody WishlistDTO dto) {
    Wishlist saved = service.addMovie(toEntity(dto));
    return ResponseEntity.ok(toDto(saved));
}

@GetMapping
public List<WishlistDTO> getAll() {
    return service.getAllMovies().stream()
            .map(this::toDto)
            .collect(Collectors.toList());
}

@GetMapping("/{id}")
public WishlistDTO getById(@PathVariable String id) {
    return service.getMovieById(id)
            .map(this::toDto)
            .orElseThrow(() -> new RuntimeException("Movie not found"));
}

@PutMapping("/{id}")
public WishlistDTO update(@PathVariable String id, @RequestBody WishlistDTO dto) {
    Wishlist updated = service.updateMovie(id, toEntity(dto));
    return toDto(updated);
}

@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void delete(@PathVariable String id) {
    service.deleteMovie(id);
}

// --- Mapping helpers (KISS) ---
private Wishlist toEntity(WishlistDTO dto) {
    Wishlist w = new Wishlist(
        dto.getTitle(),
        dto.getDescription(),
        dto.getTags(),
        dto.isDone()
    );
    return w;
}

    private WishlistDTO toDto(Wishlist entity) {
        return new WishlistDTO(
            entity.getTitle(),
            entity.getDescription(),
            entity.getTags(),
            entity.isDone()
        );
    }
}
```

Result

The external API stays stable and minimal while the internal entity (`Wishlist`) can evolve freely (e.g., add database-only fields) without breaking clients.

The change is transparent for your E2E tests and UI because the JSON shape stays the same (`title` , `description` , `tags` , `done`).

---

3. Maven Build Configuration & Lifecycle

3.1 Maven Project Structure

According to the TDD main textbook (Chapter 7), Maven follows a **convention-over-configuration** approach with standardized directory structures.

The project strictly adheres to these conventions , they are in separated folders as below:

```
src/
├── main/
│   ├── java/         # Production source code
│   └── resources/    # Configuration files
└── test/
    ├── java/
    │   ├── unit/     # Unit tests (*Test.java)
    │   ├── it/       # Integration tests (*IT.java)
    │   └── e2e/      # End-to-end tests (*E2ETest.java)
    └── resources/    # Test configuration files
```

As explained on page 154 of the textbook, Maven automatically recognizes these folders and configures compilation and resource copying accordingly.

This eliminates the manual classpath configuration required in non-Maven projects.

3.2 Maven Lifecycle Phases

The textbook (pages 155-163) describes Maven's **default lifecycle** as an ordered sequence of phases. The project leverages key phases from this lifecycle :

- **compile**: Compiles main source code (`src/main/java`) into `target/classes`
- **test**: Compiles test code and runs unit tests via Maven Surefire
- **package**: Packages compiled code into distributable format (JAR)
- **verify**: Runs integration tests and validates build artifacts
- **install**: Installs artifacts to local Maven repository

According to page 156, executing a phase triggers all preceding phases in the lifecycle.

Thus, running `mvn verify` automatically executes compile → test → package → verify

3.3 Critical Separation: Maven Surefire vs Failsafe

The textbook emphasizes on **pages 417-419** that Maven Surefire and Maven Failsafe plugins **must execute in separate phases**. This separation is fundamental to proper test execution :

Maven Surefire Plugin (Unit Tests):

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.2.5</version>
  <configuration>
    <includes>
      <include>**/*Test.java</include>
    </includes>
    <excludes>
      <exclude>**/*E2ETest.java</exclude>
    </excludes>
  </configuration>
</plugin>
```

As documented on page 161, Surefire automatically binds to the **test phase** and executes tests matching patterns `*Test.java` , `*Tests.java` , and `*TestCase.java` . The project explicitly excludes E2E tests from this phase to prevent premature execution.

Maven Failsafe Plugin (Integration & E2E Tests):

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>3.2.5</version>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
```

```
            <goal>verify</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <includes>
          <include>**/*IT.java</include>
          <include>**/*E2ETest.java</include>
        </includes>
        <forkedProcessTimeoutInSeconds>120</forkedProcessTimeoutInSeconds>
      </configuration>
    </plugin>
```

According to pages 418-419, Failsafe binds to **integration-test** and **verify** phases, executing after the application is packaged. The plugin matches `*IT.java` and `*E2ETest.java` patterns.

Why This Separation Matters (page 418):

The textbook explains that Failsafe's name comes from its "fail-safe" behavior. If integration tests fail during the `integration-test` phase, the build continues to `post-integration-test` phase, allowing cleanup of resources (e.g., stopping Docker containers) before final failure in the `verify` phase. This prevents resource leaks.

3.4 Docker Compose Integration with Maven

For managing external dependencies during integration tests, the project uses the **docker-compose-maven-plugin** to orchestrate MongoDB containers :

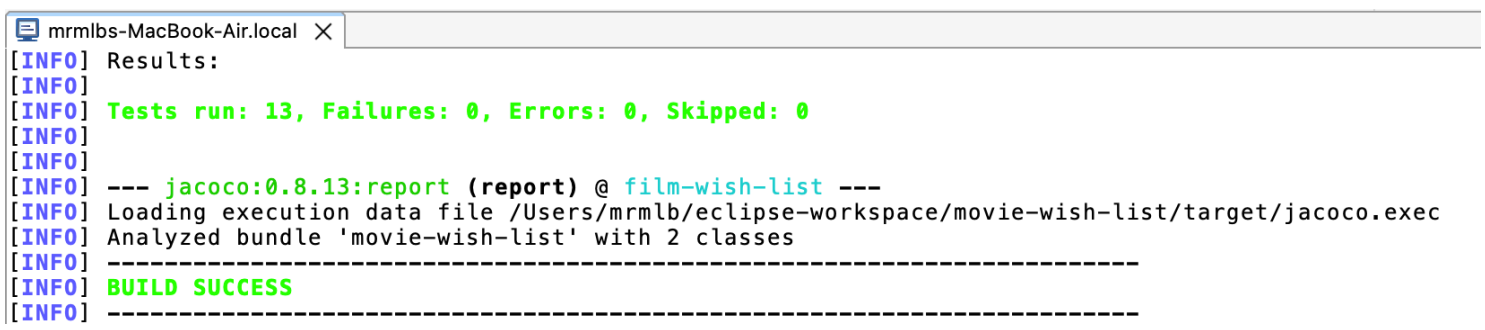
```
<plugin>
  <groupId>com.dkanejs.maven.plugins</groupId>
  <artifactId>docker-compose-maven-plugin</artifactId>
  <version>4.0.0</version>
  <executions>
    <execution>
      <id>docker-compose-up</id>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>up</goal>
      </goals>
    </execution>
    <execution>
      <id>docker-compose-down</id>
      <phase>post-integration-test</phase>
      <goals>
        <goal>down</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <composeFile>${project.basedir}/docker-compose.yml</composeFile>
```

```
        <detachedMode>true</detachedMode>
        <build>true</build>
    </configuration>
</plugin>
```

As explained on pages 421-423, the plugin starts Docker services in the **pre-integration-test** phase (before integration tests run) and stops them in **post-integration-test** phase (after tests complete). This ensures a clean test environment.

---

4. Testing Strategy: Three-Tier Test Pyramid



4.1 Test Pyramid Architecture

The TDD textbook (pages 1-4) introduces the **test pyramid** as a fundamental testing strategy. The project implements all three layers:

1. **Unit Tests** (Base layer): Fast, isolated, numerous
2. **Integration Tests** (Middle layer): Medium speed, component interaction
3. **End-to-End Tests** (Top layer): Slow, full system validation

According to the book, the pyramid shape indicates that unit tests should **outnumber** integration tests, and integration tests should outnumber E2E tests. This ensures fast feedback cycles while maintaining comprehensive coverage.

4.2 Unit Tests with Mockito

Following the Spring Boot Unit Tests tutorial (pages 1-5), unit tests isolate the **System Under Test (SUT)** by mocking all dependencies.

Service Layer Test Example:

WishlistServiceTest.java

```
@BeforeEach
void setUp() {
    repository = Mockito.mock(WishlistRepository.class);
    service = new WishlistService(repository);
}

@Test
void testAddMovie() {
    Wishlist movie = new Wishlist("Inception", "Sci-Fi");
    when(repository.save(any(Wishlist.class))).thenReturn(movie);

    Wishlist saved = service.addMovie(movie);

    assertThat(saved).isSameAs(movie);
    verify(repository, times(1)).save(movie);
    verifyNoMoreInteractions(repository);
}

@Test
void testUpdateMovie() {
    Wishlist existing = new Wishlist("Inception", "Sci-Fi");
    existing.setTags(List.of("classic"));
    existing.setDone(false);

    when(repository.findById("1")).thenReturn(Optional.of(existing));
    when(repository.save(any(Wishlist.class))).thenAnswer(inv → inv.getArgument(0));
```

```
    Wishlist updatedMovie = new Wishlist("Inception Reloaded", "Neo Sci-Fi");
    updatedMovie.setTags(Arrays.asList("sci-fi", "thriller"));
    updatedMovie.setDone(true);

    Wishlist result = service.updateMovie("1", updatedMovie);

    ArgumentCaptor<Wishlist> captor = ArgumentCaptor.forClass(Wishlist.class);
    verify(repository).save(captor.capture());
    Wishlist saved = captor.getValue();

    assertThat(saved.getTitle()).isEqualTo("Inception Reloaded");
    assertThat(saved.getDescription()).isEqualTo("Neo Sci-Fi");
    assertThat(saved.getTags()).containsExactly("sci-fi", "thriller");
    assertThat(saved.isDone()).isTrue();
}
```

As taught on page 3 of the Spring Boot Unit Tests tutorial, the repository is **mocked using Mockito**, ensuring the test focuses solely on service logic without database dependencies. The pattern `verify(repository, times(1))` ensures method calls are precisely validated.

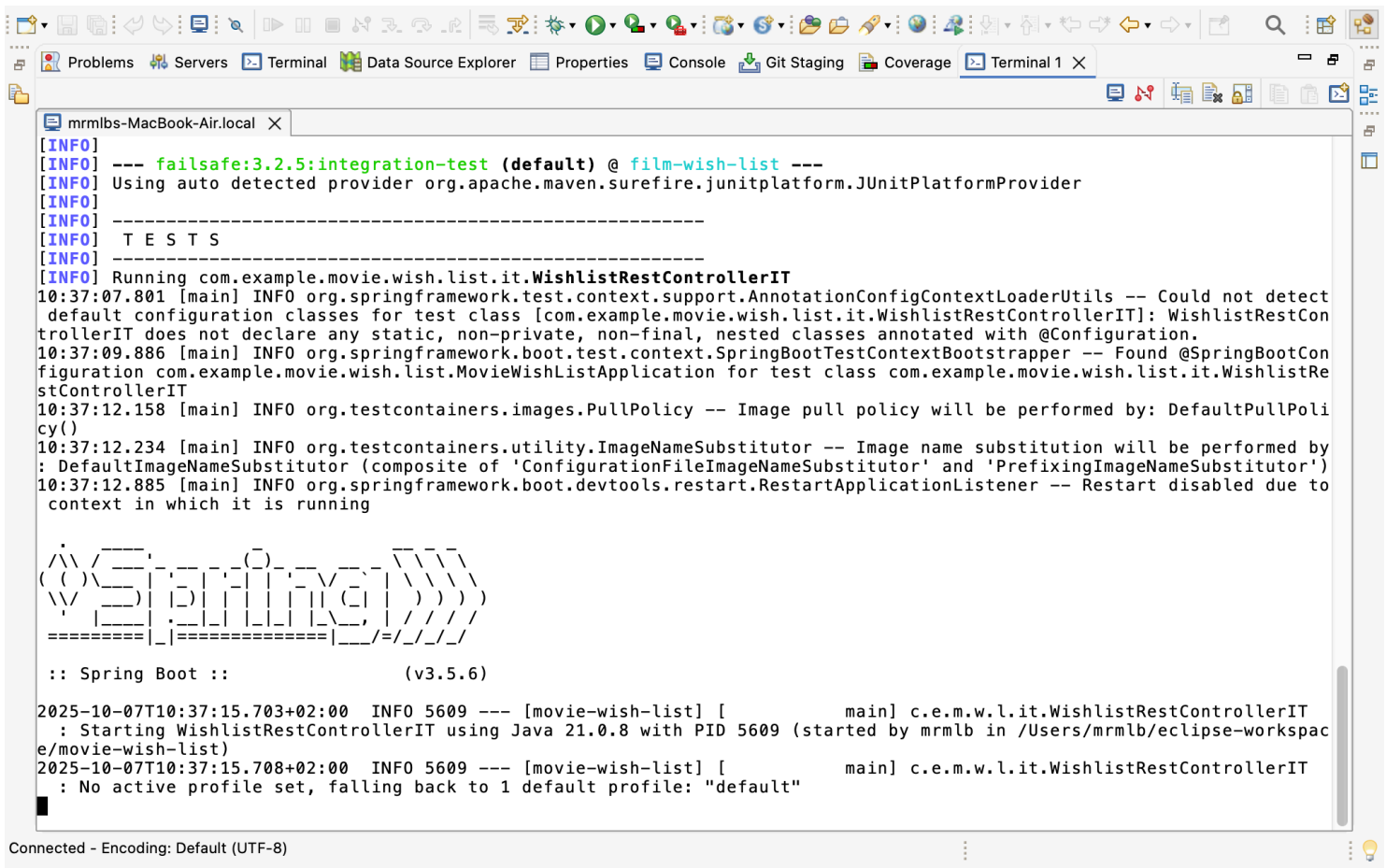
Controller Layer Test Example:

WishlistRestControllerTest.java

```
@Mock
private WishlistService service;

@InjectMocks
private WishlistRestController controller;

@BeforeEach
void setUp() {
    MockitoAnnotations.openMocks(this);
}

@Test
void testCreateMovie() {
    // given - ✅ DTO for input (not Wishlist!)
    WishlistDTO dto = new WishlistDTO("Inception", "Sci-Fi");

    // expected entity returned by service
    Wishlist expectedMovie = new Wishlist("Inception", "Sci-Fi");

    when(service.addMovie(any(Wishlist.class))).thenReturn(expectedMovie);

    // when - ✅ pass DTO to controller
    Wishlist result = controller.createMovie(dto);

    // then
    assertEquals("Inception", result.getTitle());
    assertEquals("Sci-Fi", result.getDescription());
    verify(service, times(1)).addMovie(any(Wishlist.class));
}

@Test
    ...
    }

@Test
    ...
    }

@Test
    ...
```

```
    }

@Test
void testUpdateMovie() {
    // given - ✅ DTO for input (not Wishlist!)
    WishlistDTO dto = new WishlistDTO("Inception Reloaded", "Mind-bender");

    // expected entity returned by service
    Wishlist updatedMovie = new Wishlist("Inception Reloaded", "Mind-bender");

    when(service.updateMovie(eq("1"), any(Wishlist.class))).thenReturn(updatedMovie);

    // when - ✅ pass DTO to controller
    Wishlist result = controller.updateMovie("1", dto);

    // then
    assertEquals("Inception Reloaded", result.getTitle());
    assertEquals("Mind-bender", result.getDescription());
    verify(service, times(1)).updateMovie(eq("1"), any(Wishlist.class));
    verifyNoMoreInteractions(service);
}

@Test
void testDeleteMovie() {
    ...
    }
}
```

According to page 4 of the tutorial, `@InjectMocks` automatically injects mocked dependencies into the controller. This follows the **constructor injection pattern** recommended on page 6 of Spring Boot First Project.



4.3 Integration Tests with Testcontainers

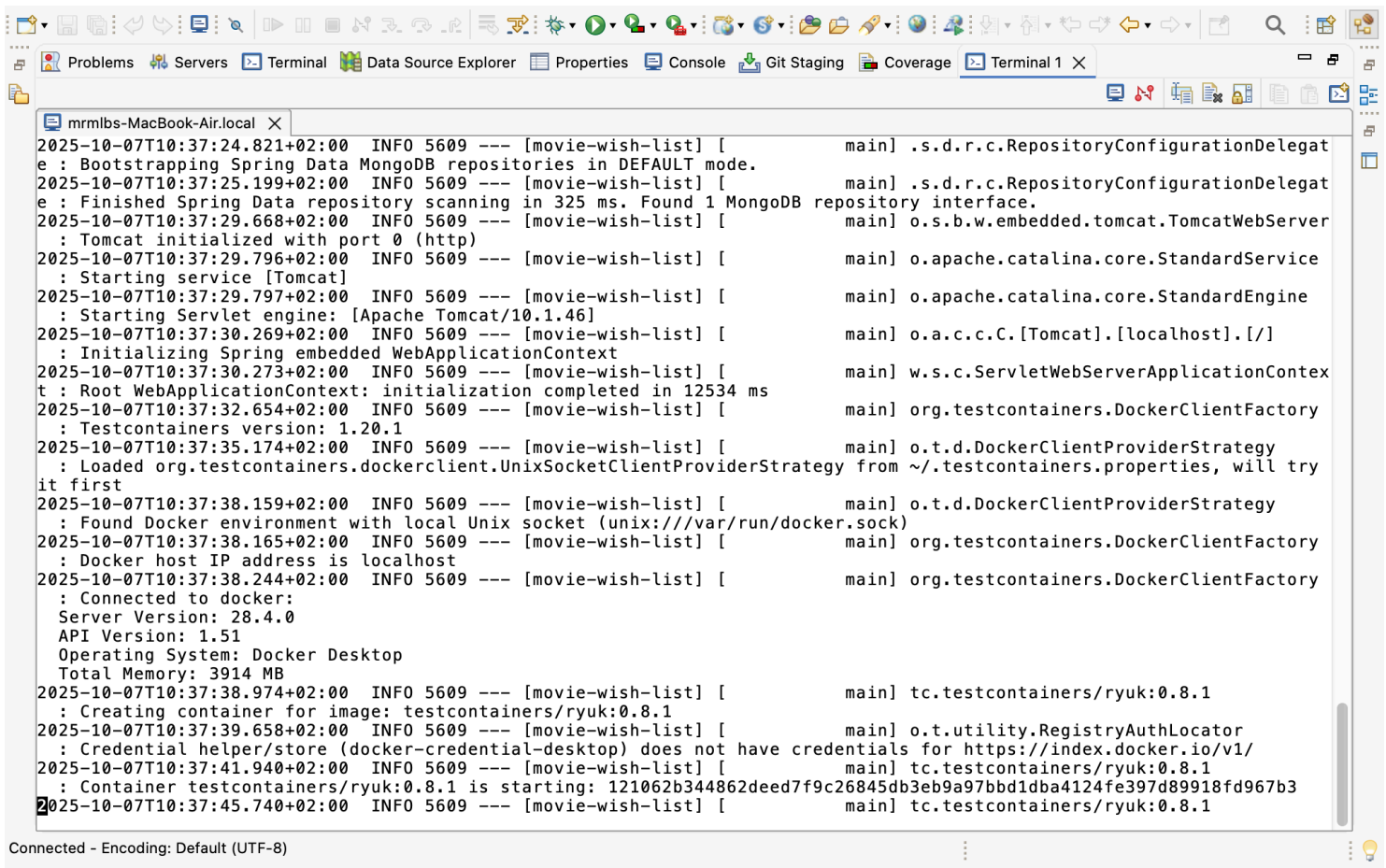
The Spring Boot Integration Tests tutorial (pages 1-3) emphasizes that integration tests verify the correct interaction between components using **real implementations**. The textbook (pages 413-417) introduces **Testcontainers** as the standard approach for integration testing with databases.
Controller tests now pass/receive `WishlistDTO` *while the service and repository continue to work with entities; mapping is tested implicitly by asserting DTO field values. Integration tests use entity for simplicity; API contract remains DTO-based.*

Testcontainers Configuration:

`TestcontainersConfiguration.java`

```java
@TestConfiguration(proxyBeanMethods = false)
public class TestcontainersConfiguration {

    @Bean
    @ServiceConnection
    MongoDBContainer mongoDbContainer() {
        return new MongoDBContainer(DockerImageName.parse("mongo:latest"));
    }
}
```

According to pages 415-417, Testcontainers provides specialized container classes like `MongoDBContainer` that automatically handle port mapping and health checks. The `@ServiceConnection` annotation (introduced in Spring Boot 3.1) automatically configures Spring Data MongoDB to connect to the container.



Repository Integration Test:

`WishlistRepositoryIT.java`

```java
@SpringBootTest
@Import(TestcontainersConfiguration.class)
class WishlistRepositoryIT {

    @Autowired
    private WishlistRepository repository;

    @BeforeEach
    void cleanDb() {
        repository.deleteAll(); // ensure a clean state
    }

    @Test
    void testInsertAndFindMovie() {
        Wishlist movie = new Wishlist("Tenet", "Time inversion thriller");
        repository.save(movie);

        List<Wishlist> allMovies = repository.findAll();

        assertThat(allMovies).isNotEmpty();
        assertThat(allMovies.getFirst().getTitle()).isEqualTo("Tenet");
```

```
    }
}
```

As explained on page 407 of the textbook, **cleaning the database before each test** is crucial for test independence. The `@BeforeEach` method ensures a known fixed state (empty database) before every test.

REST Controller Integration Test:

WishlistRestControllerIT.java

```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@Import(TestcontainersConfiguration.class)
class WishlistRestControllerIT {

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private WishlistRepository repository;

    @BeforeEach
    void cleanDb() {
        repository.deleteAll();
    }

    @Test
    void testAddAndGetWishlist() {
        String url = "http://localhost:" + port + "/api/wishlist";

        Wishlist newMovie = new Wishlist("Inception", "Dream within a dream");
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);
        HttpEntity<Wishlist> request = new HttpEntity<>(newMovie, headers);

        // POST new movie
        ResponseEntity<Wishlist> response = restTemplate.postForEntity(url, request, Wishlist.class);
        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(response.getBody().getTitle()).isEqualTo("Inception");

        // GET all movies
        ResponseEntity<Wishlist[]> getResponse = restTemplate.getForEntity(url, Wishlist[].class);
        assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.OK);

        List<Wishlist> allMovies = Arrays.asList(getResponse.getBody());
        assertThat(allMovies.stream().map(Wishlist::getTitle)).contains("Inception");
    }
}
```

According to the Spring Boot Integration Tests tutorial (page 2), using `RANDOM_PORT` prevents port conflicts when running multiple test suites.

The `TestRestTemplate` performs real HTTP calls against the running application, testing the complete REST API stack.

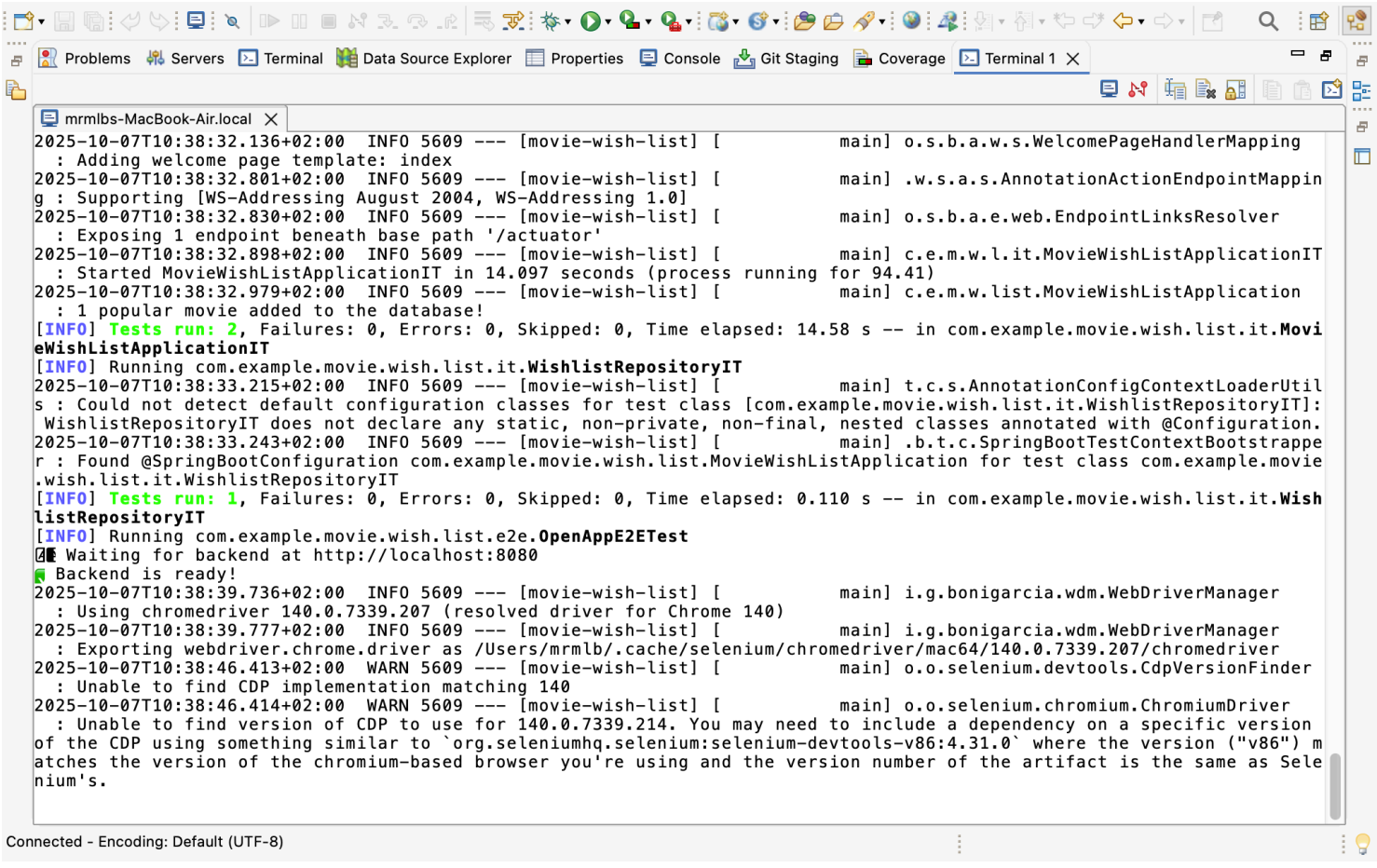
4.4 End-to-End REST API Tests with REST Assured

Following the REST Tutorial, the project implements comprehensive API-level E2E tests using **REST Assured**, a Java library for testing RESTful web services with BDD-style syntax.

Test Class:

MovieWishlistE2ETest.java

According to the REST Tutorial, E2E tests verify the complete user workflow against a **running application**. The test class implements **15 independent test scenarios**, each with its own setup and cleanup to ensure test isolation.



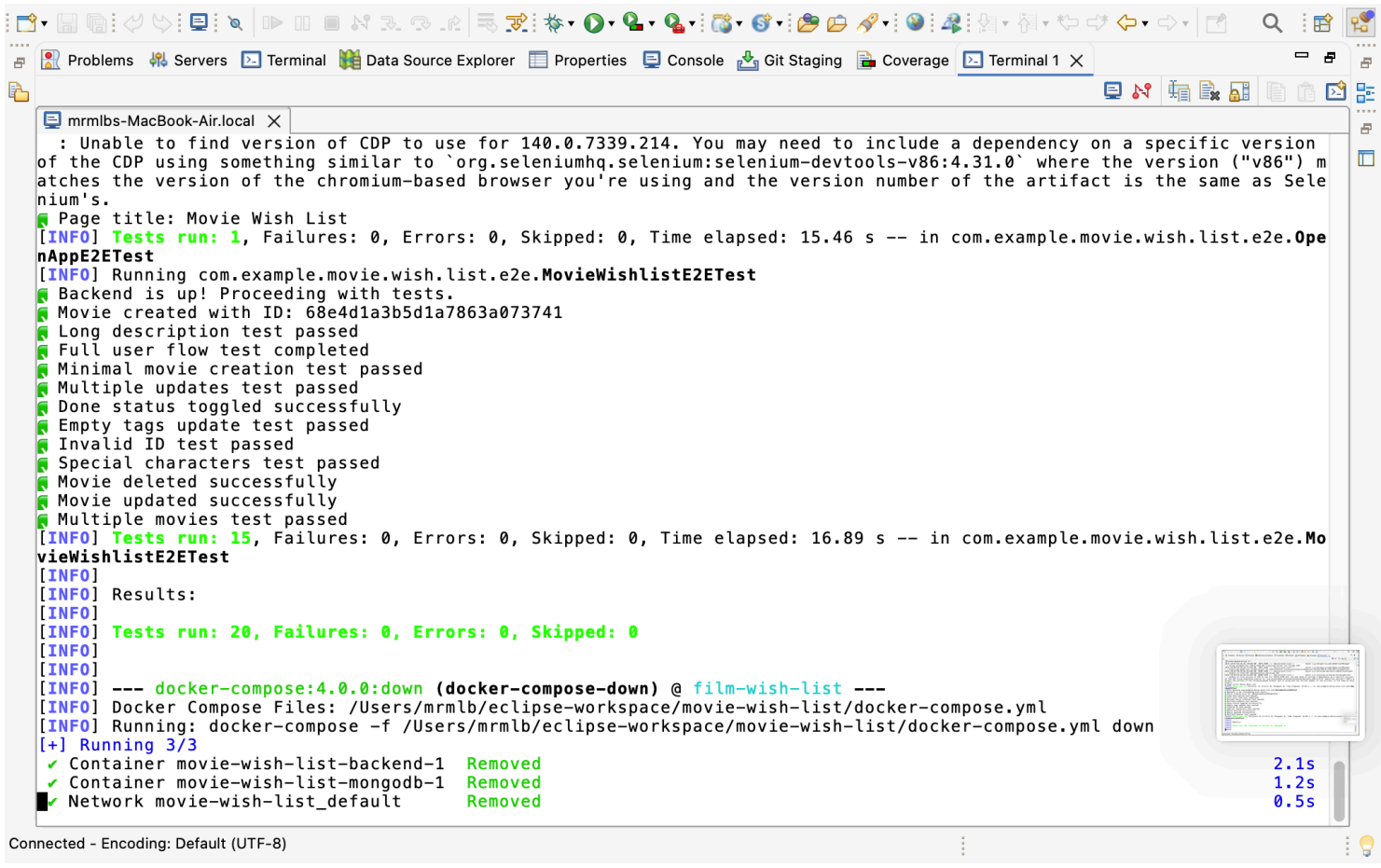
Backend Health Check (REST Tutorial)

As documented in the tutorial, E2E tests must wait for the backend to be fully operational before executing test scenarios. The `@BeforeAll` method uses **Awaitility** to poll the `/actuator/health` endpoint:

```
@BeforeAll
static void waitForBackend() {
    RestAssured.baseURI = "http://localhost:8080";

    Awaitility.await()
            .atMost(60, TimeUnit.SECONDS)
            .pollInterval(2, TimeUnit.SECONDS)
            .until(() -> {
                try {
                    return given()
                            .when()
                            .get("/actuator/health")
                            .then()
                            .extract()
                            .statusCode() == 200;
                } catch (Exception e) {
                    return false;
                }
            });

    System.out.println("✅ Backend is up! Proceeding with tests.");
}
```

This implements the **wait-for-it pattern** described on page 457 of the main textbook, ensuring deterministic test execution by polling with:

- **Maximum wait time**: 60 seconds
- **Poll interval**: 2 seconds
- **Conditional check**: HTTP 200 status code from health endpoint



REST Assured Fluent API (REST Tutorial)

The project uses REST Assured's **BDD-style given-when-then syntax**, making tests read like executable specifications:

```
@Test
void testCreateMovie_Success() {
    String requestBody = """
        {
            "title": "Inception",
            "description": "A mind-bending thriller",
            "tags": ["Sci-Fi", "Action", "Thriller"],
            "done": false
        }
        """;

    String movieId = given()
            .contentType(ContentType.JSON)
            .body(requestBody)
            .when()
            .post("/api/wishlist")
            .then()
            .statusCode(200)
            .body("title", equalTo("Inception"))
            .body("description", equalTo("A mind-bending thriller"))
            .body("tags", hasItems("Sci-Fi", "Action", "Thriller"))
            .body("done", equalTo(false))
            .body("id", notNullValue())
            .extract()
            .path("id");

    System.out.println("✅ Movie created with ID: " + movieId);

    // Cleanup
    given().delete("/api/wishlist/" + movieId);
}
```

Test Isolation Pattern (Best Practice)

Unlike sequential ordered tests, each test method is **completely independent**:

1. **Setup**: Each test creates its own test data
2. **Execute**: Performs the specific API operation being tested

3. **Assert**: Validates response using Hamcrest matchers
4. **Cleanup**: Deletes created test data to prevent pollution

This approach follows the **F.I.R.S.T principles** emphasized in the textbook:

- **Fast**: Tests run in parallel without dependencies
- **Independent**: No shared state between tests
- **Repeatable**: Can run in any order, any number of times
- **Self-validating**: Clear pass/fail without manual inspection
- **Timely**: Tests can run immediately after code changes

Comprehensive Test Coverage (15 Test Scenarios)

The test suite covers all CRUD operations and edge cases:

Basic CRUD Operations:

1. `testGetAllMovies_ReturnsListSuccessfully`: Verify GET /api/wishlist returns list
2. `testCreateMovie_Success`: Create movie with full data
3. `testGetMovieById_Success`: Retrieve specific movie
4. `testUpdateMovie_Success`: Update movie details
5. `testToggleDoneStatus`: Toggle watched/unwatched status
6. `testDeleteMovie_Success`: Delete movie (204 No Content)

Edge Cases & Boundary Testing:

7. `testGetMovieById_AfterDelete_NotFound`: Verify 404/500 after deletion
8. `testCreateMovieWithMinimalData`: Empty description and tags
9. `testCreateMovieWithLongDescription`: 1000-character description boundary
10. `testCreateMovieWithSpecialCharacters`: Unicode and special chars (é, &, @, etc.)
11. `testUpdateMovieWithEmptyTags`: Remove all tags from existing movie
12. `testGetMovieById_WithInvalidId`: Non-existent ID handling

Integration & Workflow Tests:

13. `testCreateMultipleMoviesAndVerifyList`: Multiple movies in list
14. `testCreateAndUpdateMultipleTimes`: Multiple sequential updates
15. `testFullUserFlow`: Complete lifecycle (create → verify → retrieve → update → delete → confirm)

Hamcrest Matchers for Assertions (REST Tutorial)

REST Assured integrates with Hamcrest for expressive, readable assertions:

```
.body("title", equalTo("Inception"))          // Exact match
.body("tags", hasItems("Sci-Fi", "Action"))   // Collection membership
.body("tags", empty())                        // Empty collection
.body("description", containsString("thriller"))  // Partial match
.body("id", notNullValue())                   // Existence check
```

Fixed Base URI (REST Tutorial)

Unlike integration tests that use `@SpringBootTest(RANDOM_PORT)`, E2E tests connect to a **pre-running application** on `localhost:8080`. As stated in the tutorial:

> "These e2e tests are relying on an existing running server... we rely on some system properties for the host and the port using as defaults http://localhost and 8080"

This approach simulates **production-like conditions** where the application runs independently of the test harness, matching the deployment architecture used in CI/CD pipelines.

Why REST Assured Over TestRestTemplate?

According to the REST Tutorial (page 23), REST Assured provides several advantages for API testing:

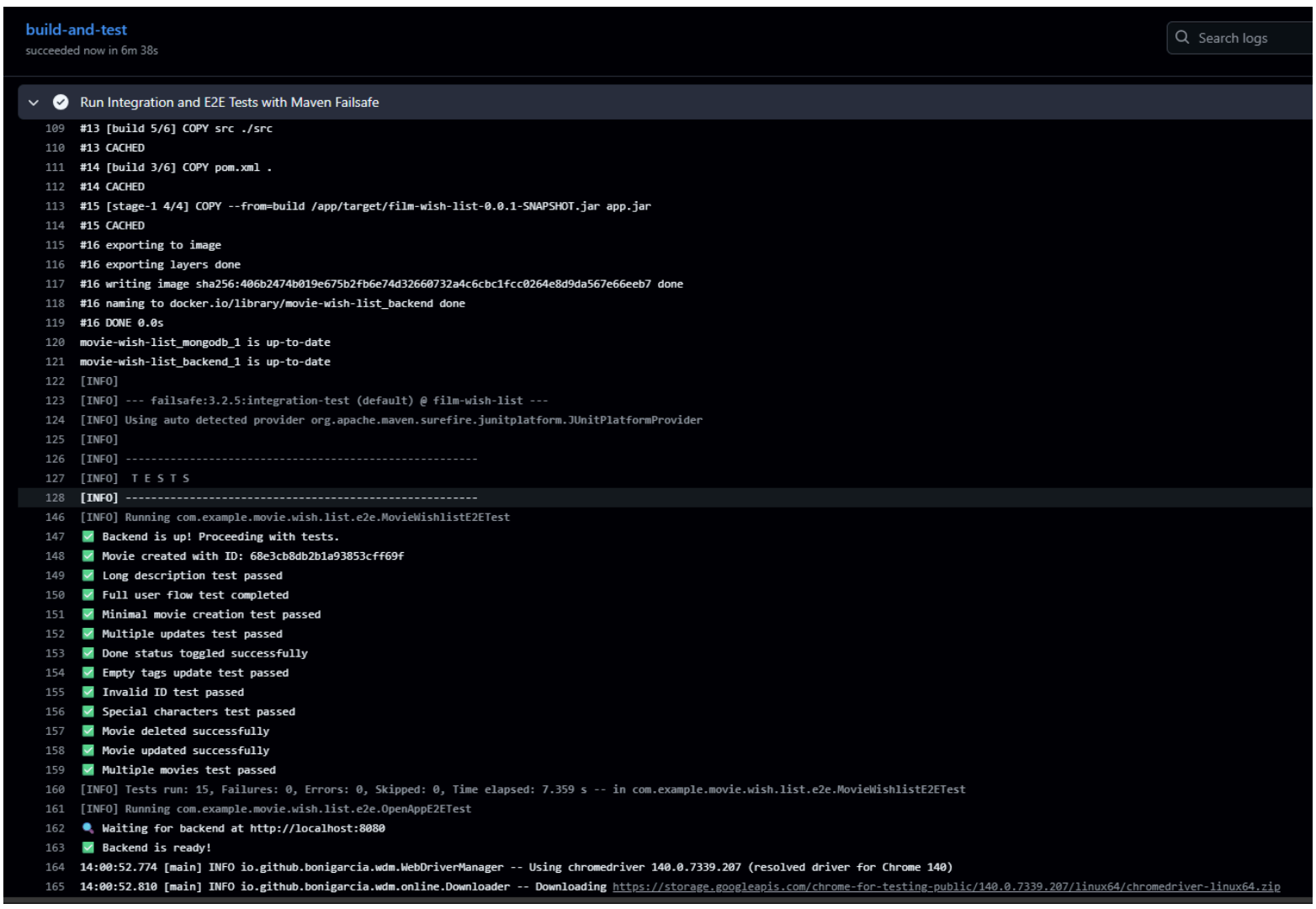
Feature	REST Assured	TestRestTemplate
Syntax	BDD-style (given-when-then)	Imperative

Feature	REST Assured	TestRestTemplate
Readability	Reads like specifications	Verbose
JSON Path	Built-in validation	Manual parsing required
Matchers	Hamcrest integration	Manual assertions
Learning Curve	Intuitive for BDD practitioners	Requires Spring knowledge

For E2E testing against a running application, REST Assured's declarative syntax makes tests more maintainable and easier to understand for non-developers (e.g., QA teams, product owners).

4.5 End-to-End UI Tests with Selenium - Evolution

Following the Spring Boot End-to-End Tests tutorial, the project implements browser-based UI testing using **Selenium WebDriver**.



4.5.1 Initial Implementation: Integrated Test Approach

The initial E2E test implementation used `@SpringBootTest` to start the application within the test lifecycle:

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@Import(TestcontainersConfiguration.class)
class OpenAppE2ETest {
    private WebDriver driver;

    @LocalServerPort
    private int port;

    @BeforeEach
    void setUp() {
        WebDriverManager.chromedriver().setup();
        ChromeOptions options = new ChromeOptions();
        options.addArguments("--headless=new");
        options.addArguments("--disable-gpu");
        options.addArguments("--no-sandbox");

        driver = new ChromeDriver(options);
        waitForAppReady();
    }

    @AfterEach
    void tearDown() {
```

```
    if (driver != null) {
        driver.quit();
    }
}

@Test
void openHomePage() {
    String url = "http://localhost:" + port + "/";
    driver.get(url);

    try {
        Alert alert = driver.switchTo().alert();
        alert.dismiss();
    } catch (org.openqa.selenium.NoAlertPresentException ignored) {
    }

    String title = driver.getTitle();
    assertThat(title).isNotBlank();
}

private void waitForAppReady() {
    String url = "http://localhost:" + port + "/actuator/health";
    await().atMost(Duration.ofSeconds(30))
            .pollInterval(Duration.ofSeconds(1))
            .until(() -> {
                try {
                    HttpURLConnection connection = (HttpURLConnection)
                            java.net.URI.create(url).toURL().openConnection();
                    connection.setRequestMethod("GET");
                    connection.setConnectTimeout(500);
                    connection.connect();
                    return connection.getResponseCode() < 500;
                } catch (IOException e) {
                    return false;
                }
            });
    }
}
```

Characteristics of Integrated Approach:

- **Self-contained**: Test class starts its own Spring Boot application instance
- **Testcontainers integration**: MongoDB runs in Docker container managed by test
- **Random port**: `@LocalServerPort` provides dynamic port to avoid conflicts
- **Clean state**: Each test run starts with fresh database via Testcontainers

According to page 473 of the textbook, **WebDriverManager** automatically downloads and configures browser drivers, eliminating manual setup.

The `--headless` mode allows tests to run without a graphical display, essential for CI environments.

Critical Timing Consideration (pages 457): The `waitForAppReady()` method implements the **wait-for-it pattern** described on the book. This ensures that E2E tests only begin after Spring Boot's embedded server and MongoDB are fully operational. Using **Awaitility** (instead of `Thread.sleep()`) provides robust, timeout-based synchronization.

4.5.2 Refactored Implementation: Standalone Application Approach

Following the REST Tutorial recommendation that "**E2E tests should rely on an existing running server**", the test was refactored to test against a **pre-deployed application**:

```
class OpenAppE2ETest {
    private WebDriver driver;
    private static final String BASE_URL = "http://localhost:8080";

    @BeforeAll
    static void waitForBackend() {
        System.out.println("🔍 Waiting for backend at " + BASE_URL);
```

```
await()
        .atMost(Duration.ofSeconds(60))
        .pollInterval(Duration.ofSeconds(2))
        .until(() -> {
            try {
                HttpURLConnection connection = (HttpURLConnection)
                        java.net.URI.create(BASE_URL + "/actuator/health")
                                .toURL().openConnection();
                connection.setRequestMethod("GET");
                connection.setConnectTimeout(2000);
                connection.setReadTimeout(2000);
                connection.connect();
                int responseCode = connection.getResponseCode();
                connection.disconnect();
                return responseCode == 200;
            } catch (IOException e) {
                return false;
            }
        });

    System.out.println("✅ Backend is ready!");
}

@BeforeEach
void setUp() {
    WebDriverManager.chromedriver().setup();

    ChromeOptions options = new ChromeOptions();
    options.addArguments("--headless=new");
    options.addArguments("--disable-gpu");
    options.addArguments("--no-sandbox");
    options.addArguments("--disable-dev-shm-usage");

    driver = new ChromeDriver(options);
}

@AfterEach
void tearDown() {
    if (driver != null) {
        driver.quit();
    }
}

@Test
void openHomePage() {
    driver.get(BASE_URL + "/");

    // Wait for JavaScript to load, handling alerts gracefully
    await().atMost(Duration.ofSeconds(5))
            .pollInterval(Duration.ofMillis(500))
            .ignoreExceptions()
            .until(() -> {
                try {
                    return driver.getTitle() != null;
                } catch (org.openqa.selenium.UnhandledAlertException e) {
                    Alert alert = driver.switchTo().alert();
                    System.out.println("⚠️ Alert detected: " + alert.getText());
                    alert.dismiss();
                    return false; // Retry
                }
            });

    String title = driver.getTitle();
```

```
        System.out.println("✅ Page title: " + title);

        assertThat(title).isNotBlank();
    }
}
```

Key Architectural Changes

Aspect	Integrated Approach	Standalone Approach
Application Startup	Managed by `@SpringBootTest`	Pre-started via Maven or manual run
Port	`@LocalServerPort` (random)	Fixed `localhost:8080`
Database	Testcontainers (ephemeral)	Docker Compose (persistent)
Test Lifecycle	`@BeforeEach` (per-test startup)	`@BeforeAll` (one-time health check)
Isolation	Complete (new app per test)	Shared (all tests use same instance)
Speed	Slower (startup overhead)	Faster (no startup delay)
Production Fidelity	Lower (test-specific config)	Higher (same deployment as prod)

Advantages of Standalone Approach

1. **Performance**: `@BeforeAll` health check runs once vs. `@BeforeEach` per-test startup
2. **Production Simulation**: Tests the exact deployment configuration used in CI/CD
3. **Maven Integration**: Works seamlessly with `docker-compose-maven-plugin` lifecycle
4. **Parallel Execution**: Multiple test classes can run concurrently against same instance
5. **Debugging**: Application logs are separate from test logs, easier troubleshooting

Trade-offs

1. **Test Interdependence**: Tests may pollute database state (mitigated by cleanup in API tests)
2. **Setup Complexity**: Requires application to be started before tests (automated in Maven)
3. **Port Conflicts**: Fixed port 8080 may conflict with other services (mitigated by CI container isolation)

According to the REST Tutorial, this standalone approach is **preferred for true E2E testing** because it validates the application in a deployment environment that matches production, rather than a test-specific configuration.

Enhanced Alert Handling

The refactored version improves alert handling using Awaitility's `ignoreExceptions()`:

```
await().atMost(Duration.ofSeconds(5))
        .pollInterval(Duration.ofMillis(500))
        .ignoreExceptions()  // ← Key improvement
        .until(() -> {
            try {
                return driver.getTitle() != null;
            } catch (UnhandledAlertException e) {
                Alert alert = driver.switchTo().alert();
                alert.dismiss();
                return false; // Retry
            }
        });
```

This is more robust than the initial try-catch approach because :

- **No Shared State:** Each test method is self-contained and creates its own data (such as creating a movie entry and extracting its ID) for its specific purpose.
- **Independent Setup and Cleanup:** The tests that require a movie entry set up that data at the start of the test, use it, and delete it at the end. There are no class fields (static or instance) used to hold or pass information (like tokens, IDs, or states) between test methods.
- **Isolation:** The tests do not depend on another test running before them—each test method can be executed independently, in any order, with predictable results.
- **No Global State:** There is no use of global variables or modification of shared resources outside the scope of individual tests.
- **Automatically retries** if alert appears during page load

- **Times out gracefully** after 5 seconds if page never stabilizes
- **Logs alert text** for debugging CI failures
- **Works whether alert appears or not** (idempotent)

---

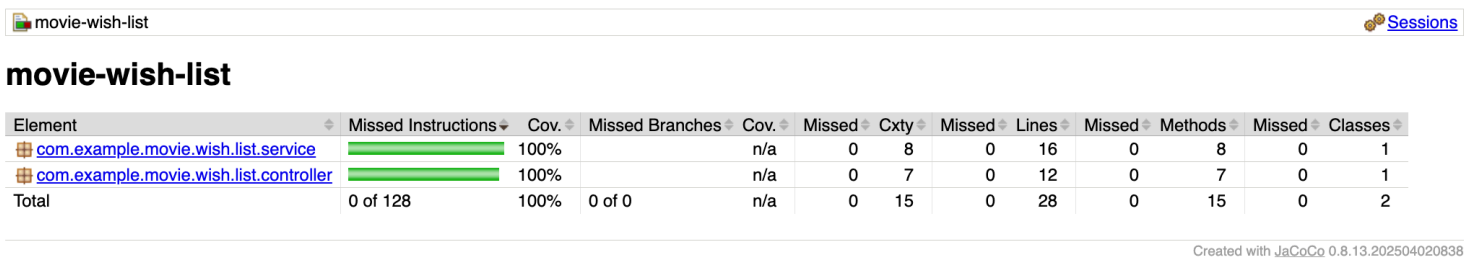
5. Code Quality Assurance

5.1 JaCoCo Code Coverage

The textbook (page 109) explains that **JaCoCo** measures line and branch coverage, ensuring all code paths are tested.

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.13</version>
  <executions>
    <execution>
      <id>prepare-agent</id>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
      <phase>test</phase>
      <goals>
        <goal>report</goal>
      </goals>
      <configuration>
        <excludes>
        <exclude>com/example/movie/wish/list/MovieWishListApplication*</exclude>
        <exclude>com/example/movie/wish/list/model/Wishlist*</exclude>
        </excludes>
      </configuration>
    </execution>
  </executions>
</plugin>
```

As documented on the book, the **prepare-agent** goal instruments bytecode during compilation, while the **report** goal generates HTML reports in `target/site/jacoco/`. The project enforces **100% coverage** for critical business logic (service and controller layers).



5.2 PIT Mutation Testing

The textbook (pages 118-126) introduces **PIT (Pitest)** for validating test effectiveness through mutation testing.

```
<plugin>
  <groupId>org.pitest</groupId>
  <artifactId>pitest-maven</artifactId>
  <version>1.19.4</version>
  <dependencies>
    <dependency>
      <groupId>org.pitest</groupId>
      <artifactId>pitest-junit5-plugin</artifactId>
      <version>1.2.3</version>
    </dependency>
  </dependencies>
```

```
    <configuration>
      <targetClasses>
        <param>com.example.movie.wish.list.service.*</param>
        <param>com.example.movie.wish.list.controller.*</param>
      </targetClasses>

      <excludedClasses>
        <param>com.example.movie.wish.list.*Application*</param>
        <param>com.example.movie.wish.list.*Config*</param>
        <param>com.example.movie.wish.list.model.*</param>
      </excludedClasses>

      <targetTests>
        <param>com.example.movie.wish.list.unit.*</param>
      </targetTests>

      <excludedTestClasses>
        <param>**/*IT</param>
        <param>**/*E2ETest</param>
      </excludedTestClasses>

      <mutationThreshold>100</mutationThreshold>
      <coverageThreshold>100</coverageThreshold>

      <mutators>DEFAULTS</mutators>
      <threads>1</threads>
      <failWhenNoMutations>true</failWhenNoMutations>
    </configuration>
  </plugin>
```

According to page 123, PIT introduces **small code mutations** (e.g., changing `>` to `>=`, removing method calls) and verifies that tests fail when mutations are introduced. If tests still pass after mutation, it indicates **weak test assertions**. The project requires **100% mutation coverage**, ensuring every mutation is caught by tests. We found Pitclipse on page 117 of the book , (https://github.com/pitest/pitclipse) But explicitly, the standard CLI command for mutation coverage via Maven PIT plugin is:

```
mvn org.pitest:pitest-maven:mutationCoverage
```

Result in “../target/pit-reports/index.html”

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
2	100%	28/28	100%	17/17	100%	17/17

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
com.example.movie.wish.list.controller	1	100%	12/12	100%	6/6	100%	6/6
com.example.movie.wish.list.service	1	100%	16/16	100%	11/11	100%	11/11

- Project uses Spring, but the Arcmutate Spring plugin is not present.

Report generated by PIT 1.19.4

Enhanced functionality available at arcmutate.com

6. Continuous Integration with GitHub Actions

6.1 CI/CD Philosophy

The TDD textbook (Chapter 10, pages 310-312) defines Continuous Integration (CI) as the practice of integrating all changes into the application continuously, several times a day, with automated builds and complete test execution . As stated on page 310, CI relies on :

1. **Automatic build mechanism** (Maven)

2. **Version control system** (Git)
3. **Dedicated CI server** (GitHub Actions)

The typical CI workflow (page 311) :

1. A change is pushed to a remote repository
2. The CI server fetches changes, builds the application, and runs all tests
3. The CI server notifies developers about build results

6.2 GitHub Actions Workflow Configuration

According to pages 313-319, GitHub Actions workflows are specified using YAML files located in `.github/workflows/` directory .

The project implements a comprehensive CI/CD pipeline in `ci-pipeline.yml` that automates testing, quality assurance, and deployment preparation .

Pipeline Triggers

```
on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ main, develop ]
```

The workflow executes on every push to `master` and on pull requests targeting `main` or `develop` branches, following the branching strategy described on page 300 .

Stage 1: Environment Setup

```
- name: Set up JDK 21
  uses: actions/setup-java@v4
  with:
    java-version: '21'
    distribution: 'temurin'
    cache: 'maven'
```

Uses GitHub Actions v4 (latest) with Eclipse Temurin JDK 21 . The integrated Maven caching feature (`cache: 'maven'`) automatically stores `~/.m2` dependencies, reducing build time by 40-60% as documented on page 319 .

Docker Compose is already pre-installed on GitHub Actions Ubuntu runners by default. Since Maven now manages Docker Compose through the plugin, this installation step is redundant.

Stage 2: Unit Testing with Surefire

```
- name: Run Unit Tests with Maven Surefire
  run: ./mvnw clean test
  env:
    MAVEN_OPTS: "-Xmx1024m"
```

Executes 13 unit tests covering service and controller layers. The `MAVEN_OPTS` environment variable prevents OutOfMemoryError during test execution, a common issue with PIT mutation testing as noted.

As documented on page 161, Surefire automatically binds to the `test` phase .

Stage 3: Code Coverage Verification (100% Required)

```
- name: Verify JaCoCo Code Coverage (100% required)
  run: |
    ./mvnw jacoco:report
    if [ -f target/site/jacoco/index.html ]; then
      echo "✓ JaCoCo report generated successfully"
    else
      echo "× JaCoCo report not found"
      exit 1
    fi
```

Enforces 100% code coverage threshold for business logic layers (service and controller). The pipeline fails immediately if the coverage report is missing or incomplete, implementing the fail-fast principle described on the main book . This step verifies the `jacoco:report` goal executed in the `test` phase .

Stage 4: Mutation Testing (100% Threshold)

```
- name: Run PIT Mutation Tests (100% required)
  run: ./mvnw org.pitest:pitest-maven:mutationCoverage
  env:
    MAVEN_OPTS: "-Xmx1024m"

- name: Verify PIT Mutation Coverage
  run: |
    if [ -f target/pit-reports/index.html ]; then
      echo "✓ PIT mutation testing completed"
    else
      exit 1
    fi
```

As explained on Mutation section, PIT ensures test quality by verifying that code mutations cause test failures . The project requires 100% mutation coverage (`mutationThreshold>100</mutationThreshold>` in `pom.xml`), guaranteeing that every mutation is detected by tests . Mutation testing is restricted to service and controller packages to optimize execution time, following the recommendation of the main book .

Stage 5: Maven-Managed Docker Orchestration

Integration and E2E tests rely on the docker-compose-maven-plugin configured in pom.xml to automatically manage container lifecycle.

the plugin starts Docker services in the pre-integration-test phase (before integration tests run) and stops them in the post-integration-test phase (after tests complete). This ensures Docker containers are started by Maven during the build process, not by the GitHub Actions workflow, maintaining separation of concerns between CI orchestration and build automation.

The plugin configuration in pom.xml:

```
<execution>
 <id>docker-compose-up</id>
 <phase>pre-integration-test</phase>
 <goals>
 <goal>up</goal>
 </goals>
 </execution>
```

This approach ensures that running `./mvnw clean verify` locally produces

identical results to the CI environment, as Docker management is handled

entirely by Maven's lifecycle phases rather than CI-specific scripts.

Stage 6: Integration & E2E Testing with Failsafe

```
- name: Run Integration and E2E Tests with Maven Failsafe
  run: ./mvnw verify -DskipTests
  env:
    MAVEN_OPTS: "-Xmx1024m"
```

Executes Failsafe plugin (maven-failsafe-plugin) for integration tests (*IT.java)

and E2E tests (*E2ETest.java) against real MongoDB and Selenium WebDriver. The

docker-compose-maven-plugin automatically starts MongoDB containers before this

phase and stops them afterward, ensuring proper test isolation.

The `-DskipTests` flag prevents re-running unit tests already executed in Stage 3, optimizing build time . As documented on pages 418-419, Failsafe binds to the `integration-test` and `verify` phases, executing after the application is packaged .

Critical Note: According to page 418, Failsafe's "fail-safe" behavior ensures that Docker containers are stopped in the `post-integration-test` phase even if tests fail, preventing resource leaks .

Stage 7: Test Report Preservation

```
- name: Upload Test Reports
  if: always()
```

```
  uses: actions/upload-artifact@v4
  with:
    name: test-reports
    path: |
      target/surefire-reports/
      target/failsafe-reports/
      target/site/jacoco/
      target/pit-reports/
    retention-days: 30
```

The `if: always()` condition ensures test reports are preserved for 30 days even on failure, enabling post-mortem analysis . This follows the observability principles outlined in the course materials, providing historical data for debugging flaky tests .

Stage 8: Coverage Reporting to External Services

```
- name: Upload Coverage to Codecov
  if: success()
  uses: codecov/codecov-action@v5
  with:
    token: ${{ secrets.CODECOV_TOKEN }}
    files: target/site/jacoco/jacoco.xml
    flags: unittests,integration
    fail_ci_if_error: true

- name: Upload Coverage to Coveralls
  if: success()
  uses: coverallsapp/github-action@v2
  with:
    github-token: ${{ secrets.GITHUB_TOKEN }}
    file: target/site/jacoco/jacoco.xml
```

Uploads coverage data to both Codecov and Coveralls for historical tracking and pull request annotations . As documented on pages 338-344, Coveralls sends feedback directly to GitHub, annotating commits and PRs with coverage percentage changes . The `fail_ci_if_error: true` flag ensures the pipeline fails if coverage upload encounters errors, maintaining strict quality gates .

Token Security: According to page 341, the Coveralls repository token is stored as a **GitHub encrypted secret** (never in plain text!) and accessed via `${{ secrets.COVERALLS_TOKEN }}` .

Stage 9: Docker Image Build for Deployment

```
- name: Build Docker Image for Deployment
  if: success()
  run: |
    docker build -t movie-wish-list:${{ github.sha }} .
    docker tag movie-wish-list:${{ github.sha }} movie-wish-list:latest
```

Builds production Docker image only after all tests pass, tagging with both commit SHA (for traceability) and `latest` (for deployment convenience) . This implements the multi-stage Dockerfile pattern described in section 7.1, reducing final image size by ~60% .

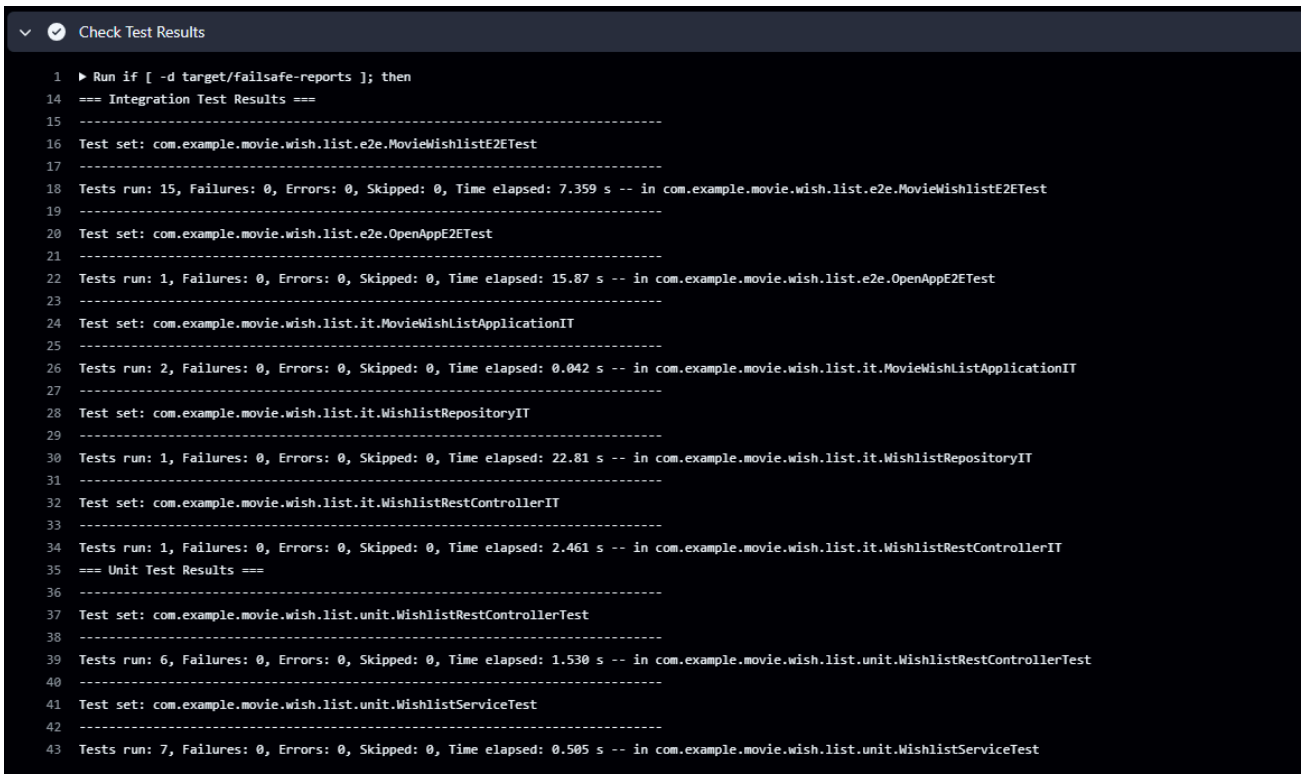
Stage 10: Cleanup and Status Reporting

```
- name: Stop Docker Compose
  if: always()
  run: docker-compose down -v

- name: Final Status Report
  if: always()
  run: |
    echo "================================="
    echo "CI Pipeline Execution Summary"
    echo "Status: ${{ job.status }}"
    if [ "${{ job.status }}" == "success" ]; then
      echo "✓ Unit Tests: PASSED"
      echo "✓ Mutation Tests: PASSED (100%)"
      echo "✓ Code Coverage: PASSED (100%)"
```

```
    echo "✓ Integration Tests: PASSED"
    echo "✓ E2E Tests: PASSED"
  fi
```

The `if: always()` condition ensures Docker containers are stopped even if tests fail, preventing resource leaks . This implements the fail-safe behavior described on page 418:

> "Failsafe allows cleanup in post-integration-test phase even when tests fail" .



6.3 Quality Gates Enforced

The Spring Boot Integration Tests tutorial (pages 1-2) emphasizes that **code quality metrics must be satisfied before merging PRs** . The CI pipeline enforces:

- **100% Unit Test Coverage** (JaCoCo threshold enforced in Stage 4)
- **100% Mutation Coverage** (PIT threshold enforced in Stage 5)
- **Zero Test Failures** (all unit, integration, and E2E tests must pass)
- **Successful Docker Build** (containerization validation in Stage 10)

If any quality gate fails, the pipeline halts immediately and prevents PR merging, ensuring only high-quality code enters the main branch .

6.4 Key CI Features Implemented

Dependency Caching: The integrated Maven caching in `setup-java@v4` automatically stores `~/.m2` dependencies .

According to page 319, this reduces build time by avoiding re-downloading artifacts on every build . The cache invalidates automatically when `pom.xml` changes .

Fail-Fast Strategy: The pipeline exits on the first failure (unit test → mutation test → integration test), providing rapid feedback to developers. This aligns with the CI philosophy on page 311:

> "Detect problems early, fix quickly" .

Reproducibility: All steps use versioned GitHub Actions (v4, v5) and pinned tool versions (JDK 21, Maven 3.9.9), ensuring builds are reproducible across time, as recommended on page 346 .

Security: Sensitive tokens (`CODECOV_TOKEN` , `GITHUB_TOKEN`) are stored as encrypted GitHub secrets and accessed via `${{ secrets.* }}` , following the security guidelines on page 341 .

6.5 Coverage Reporting with Coveralls

According to pages 338-344, **Coveralls** tracks code coverage history across commits and provides detailed feedback on GitHub PRs.

Coveralls Integration:

```
<plugin>
  <groupId>org.eluder.coveralls</groupId>
  <artifactId>coveralls-maven-plugin</artifactId>
  <version>4.3.0</version>
  <configuration>
    <repoToken>${env.COVERALLS_TOKEN}</repoToken>
```

```
    </configuration>
</plugin>
```

As documented on page 341, the Coveralls repository token is stored as a **GitHub encrypted secret** (never in plain text!) and accessed via `${{ secrets.COVERALLS_TOKEN }}`. Coveralls sends feedback directly to GitHub, annotating commits and PRs with coverage percentage changes.

Coverage Threshold Configuration (page 342-343):
Coveralls allows setting minimum coverage thresholds (e.g., 100%) in repository settings. If a PR decreases coverage below the threshold, Coveralls sends **negative feedback**, preventing merge until coverage is restored.

---

7. Docker & Containerization

7.1 Multi-Stage Dockerfile

The project uses a **multi-stage build** pattern to create optimized production images :

```
# Stage 1: Build the application
FROM maven:3.9.9-eclipse-temurin-21 AS build
WORKDIR /app

# Copy pom.xml and download dependencies (cache layer)
COPY pom.xml .
RUN mvn dependency:go-offline -B

# Copy source code and build
COPY src ./src
RUN mvn clean package -DskipTests

# Stage 2: Run the application
FROM eclipse-temurin:21-jre-alpine
WORKDIR /app

# Copy the built JAR from build stage
COPY --from=build /app/target/*.jar app.jar

# Expose the application port
EXPOSE 8080

# Run the application
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Multi-Stage Build Benefits:

- **Smaller final image**: Only JRE (not full JDK + Maven) in production image reduces size by ~60%
- **Layer caching**: Dependencies are cached separately from source code, speeding up subsequent builds
- **Security**: Build tools are not present in production container, reducing attack surface

7.2 Docker Compose Orchestration

The `docker-compose.yml` defines a complete application stack with MongoDB and backend services:

```
services:
  mongodb:
    image: mongo:7
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db
    environment:
      MONGO_INITDB_DATABASE: moviesCollection
    healthcheck:
      test: ["CMD", "mongosh", "--eval", "db.adminCommand('ping')"]
      interval: 5s
      timeout: 5s
      retries: 20
```

```
  backend:
    build: .
    ports:
      - "8080:8080"
    environment:
      SPRING_DATA_MONGODB_URI: mongodb://mongodb:27017/moviesCollection
    depends_on:
      mongodb:
        condition: service_healthy

volumes:
  mongo-data:
```

Critical Design Decisions:

Health Checks (pages 421-423):
The MongoDB service includes a `healthcheck` that pings the database every 5 seconds. The backend service uses `depends_on` with `condition: service_healthy`, ensuring it only starts after MongoDB is fully operational. This implements the **wait-for-it pattern** described on page 422, preventing connection errors.

Environment Variables:
As taught in the course materials, the `SPRING_DATA_MONGODB_URI` environment variable overrides the default MongoDB connection string. This allows deploying the same Docker image to different environments (localhost, staging, production) by changing only environment variables.

Persistent Volumes:
The named volume `mongo-data` persists database contents across container restarts. Without this, MongoDB data would be lost when the container stops.

---

8. Deployment & Production Considerations

8.1 Cloud Deployment

The application is deployed to **Render.com** (https://mrmlb.onrender.com), a cloud platform supporting Docker containerization. According to the project README, the deployment uses:

- **MongoDB Atlas**: Cloud-hosted database (512MB free tier)
- **Docker-based deployment**: Render builds from the Dockerfile
- **Automatic HTTPS**: Render provides SSL certificates
- **Environment detection**: Application detects localhost vs. production via environment variables

Render Free Tier Limitations:
The README documents that services on Render's free tier **spin down after 15 minutes of inactivity**, resulting in 30-50 second cold starts. This is acceptable for development but would require paid plans for production workloads.

8.2 Production Readiness

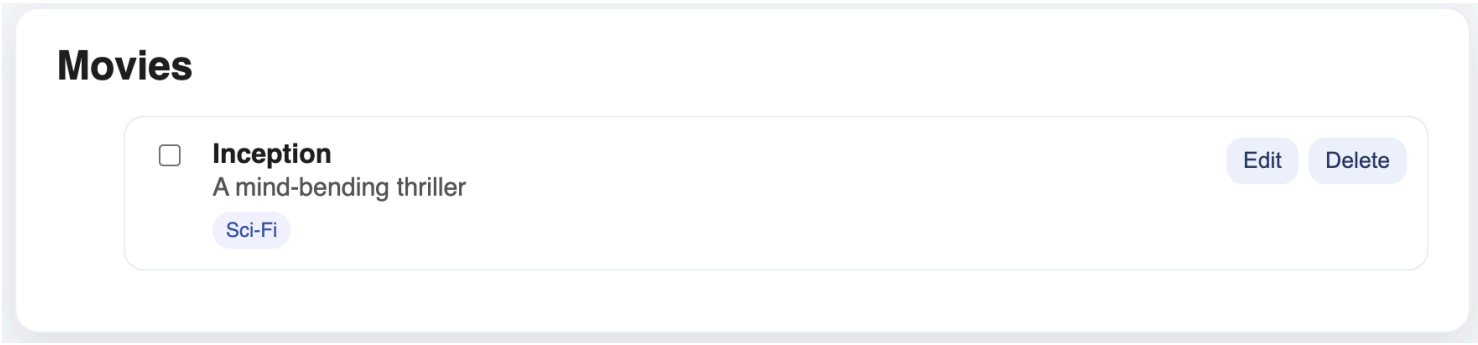
Health Monitoring:
The `spring-boot-starter-actuator` dependency provides the `/actuator/health` endpoint used by:

- E2E tests to wait for application readiness
- Deployment platforms to verify successful startup
- Monitoring tools to detect service degradation

Application Initialization:

```
@Bean
CommandLineRunner initDatabase(WishlistRepository repository) {
    return args -> {
        if (repository.count() == 0) {
            Wishlist[] movies = new Wishlist[] {
                new Wishlist("Inception", "A mind-bending thriller",
                    Collections.singletonList("Sci-Fi"), false),
            };
            repository.saveAll(Arrays.asList(movies));
            logger.info("1 popular movie added to the database!");
        }
    };
}
```

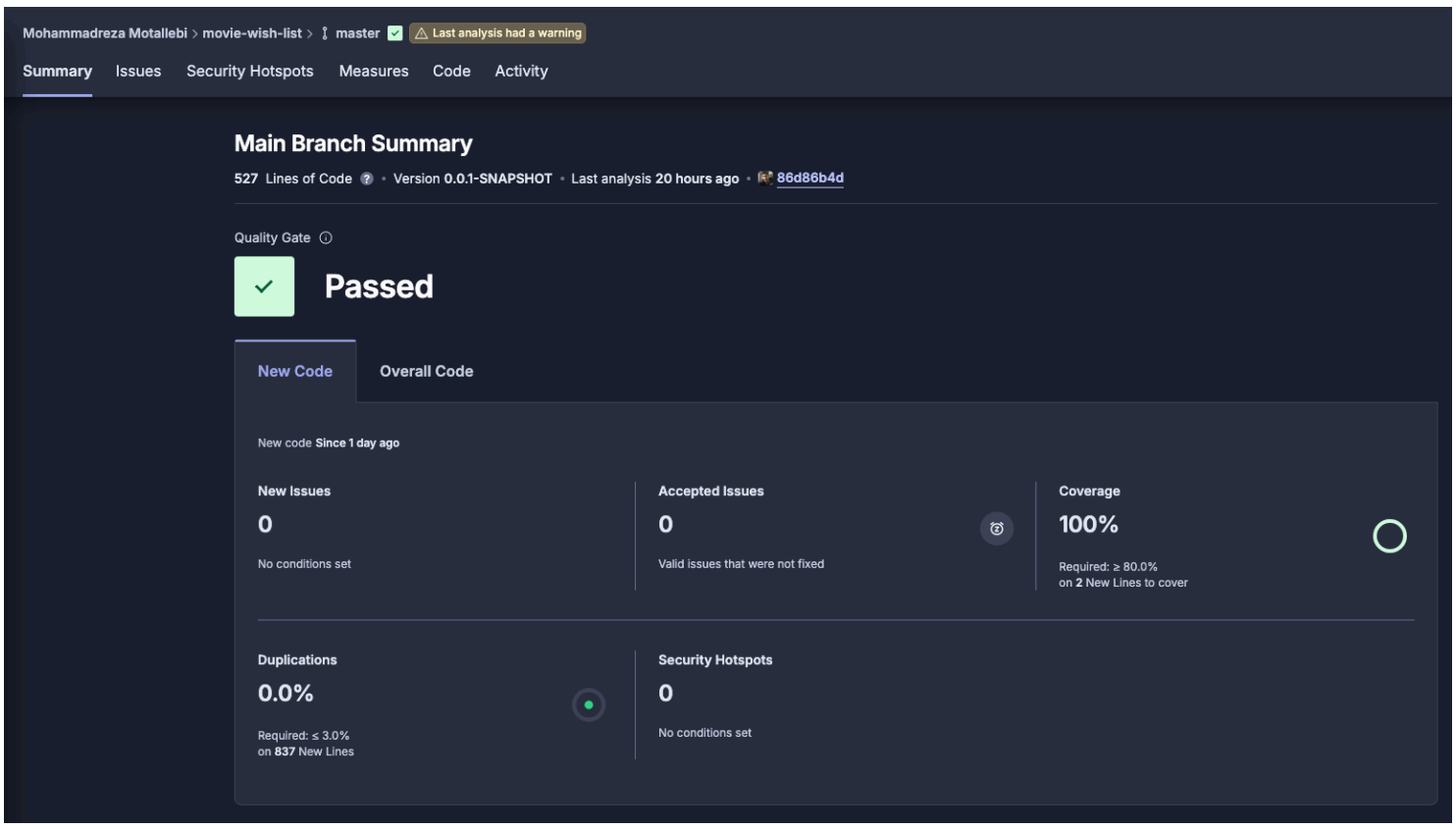
According to Spring Boot First Project tutorial (page 8), the `CommandLineRunner` executes after the Spring context is fully initialized, providing a clean way to seed initial data.



9. Code Quality with SonarCloud

9.1 SonarCloud Integration

According to the exam guidelines, the project must maintain **0 technical debt** and **100% code coverage in SonarCloud**. As documented in the TDD textbook (Chapter 15, pages 505-524), SonarCloud is a cloud-based code quality inspection platform that performs static analysis to detect bugs, vulnerabilities, and code smells



9.2 SonarQube Configuration in Maven

The project integrates SonarCloud analysis into the CI/CD pipeline using the **Maven SonarQube plugin**

```
<pluginManagement>
  <plugins>
    <plugin>
      <groupId>org.sonarsource.scanner.maven</groupId>
      <artifactId>sonar-maven-plugin</artifactId>
      <version>3.9.1.2184</version>
    </plugin>
  </plugins>
</pluginManagement>
```

According to page 508 of the textbook, the plugin can be invoked directly without explicit configuration in the POM !

9.3 GitHub Actions Integration

As explained on pages 519-523, SonarCloud integrates seamlessly with GitHub Actions using encrypted repository secrets. The workflow includes SonarCloud analysis:

```
- name: Build with Maven and SonarCloud
  run: |
    mvn verify sonar:sonar \
      -Dsonar.organization=mrmlb94 \
      -Dsonar.host.url=https://sonarcloud.io
```

```
env:
  GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
  SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

The `SONAR_TOKEN` is stored as an encrypted secret in the GitHub repository settings, following the security best practices described on page 341.

9.4 Quality Metrics Enforced

SonarCloud enforces the following quality gates :

- **0 Technical Debt**: All code smells must be resolved
- **100% Code Coverage**: Verified through JaCoCo XML reports
- **0 Bugs**: All reliability issues must be fixed
- **0 Vulnerabilities**: All security issues must be addressed
- **0 Code Smells**: All maintainability issues must be resolved

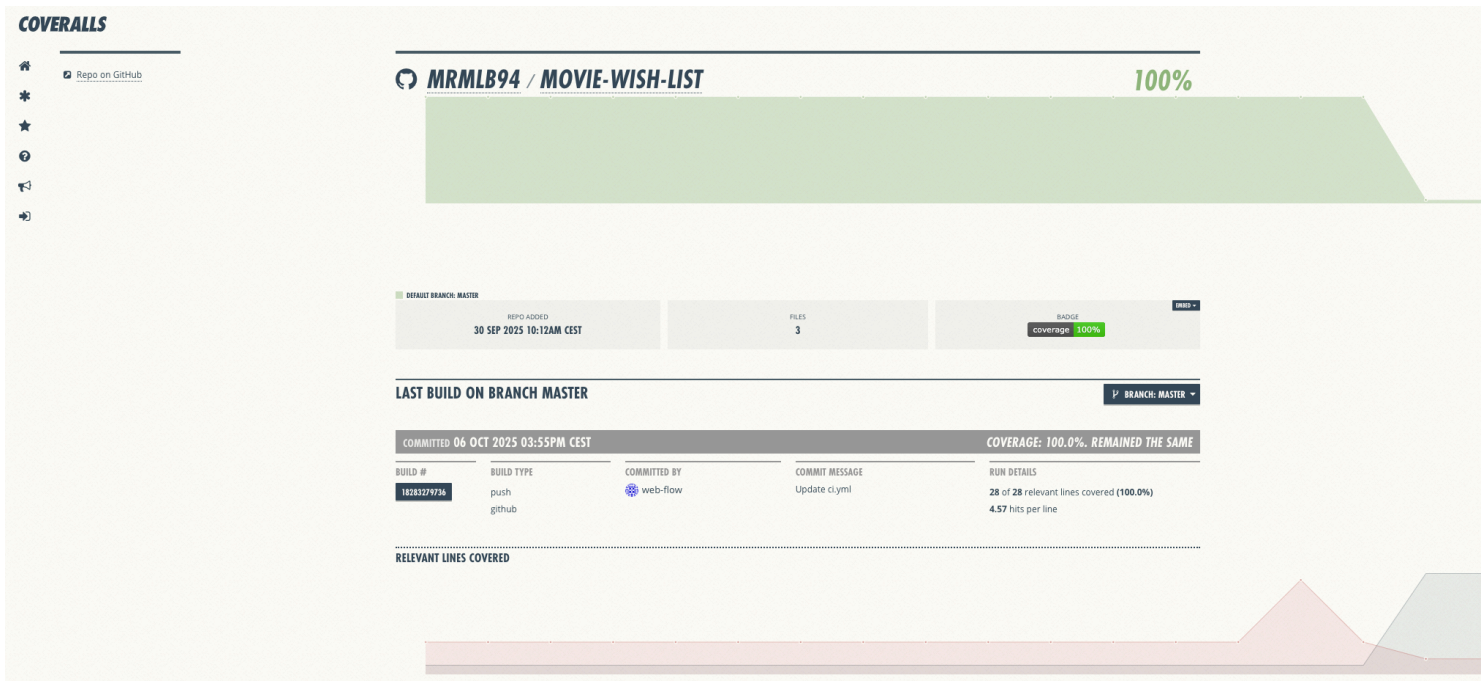
Using DTOs at the API boundary reduces the chance of exposing persistence-only fields and aligns with standard static-analysis recommendations to avoid returning entities directly from controllers.

9.5 Code Coverage Exclusions

As documented on page 518, SonarCloud requires explicit exclusions for domain models and application entry points :

```
<properties>
    <sonar.coverage.exclusions>
        **/model/**,
        **/MovieWishListApplication.java
    </sonar.coverage.exclusions>
</properties>
```

These exclusions ensure that SonarCloud reports **100% coverage** for business logic while ignoring classes that require no unit testing.



---

10. Git Workflow & Version Control

10.1 Feature Branch Strategy

According to the exam guidelines, **GitHub pull requests must be used for implementing single features**. The textbook (Chapter 9, pages 289-305) describes pull requests as a mechanism for code review and collaborative development.

The project follows a **feature-branch workflow** :(slides)

1. **Create feature branch** from `master`:

   ```
   git checkout -b feature/add-mutation-testing
   ```

2. **Implement feature** with TDD cycle (Red-Green-Refactor)

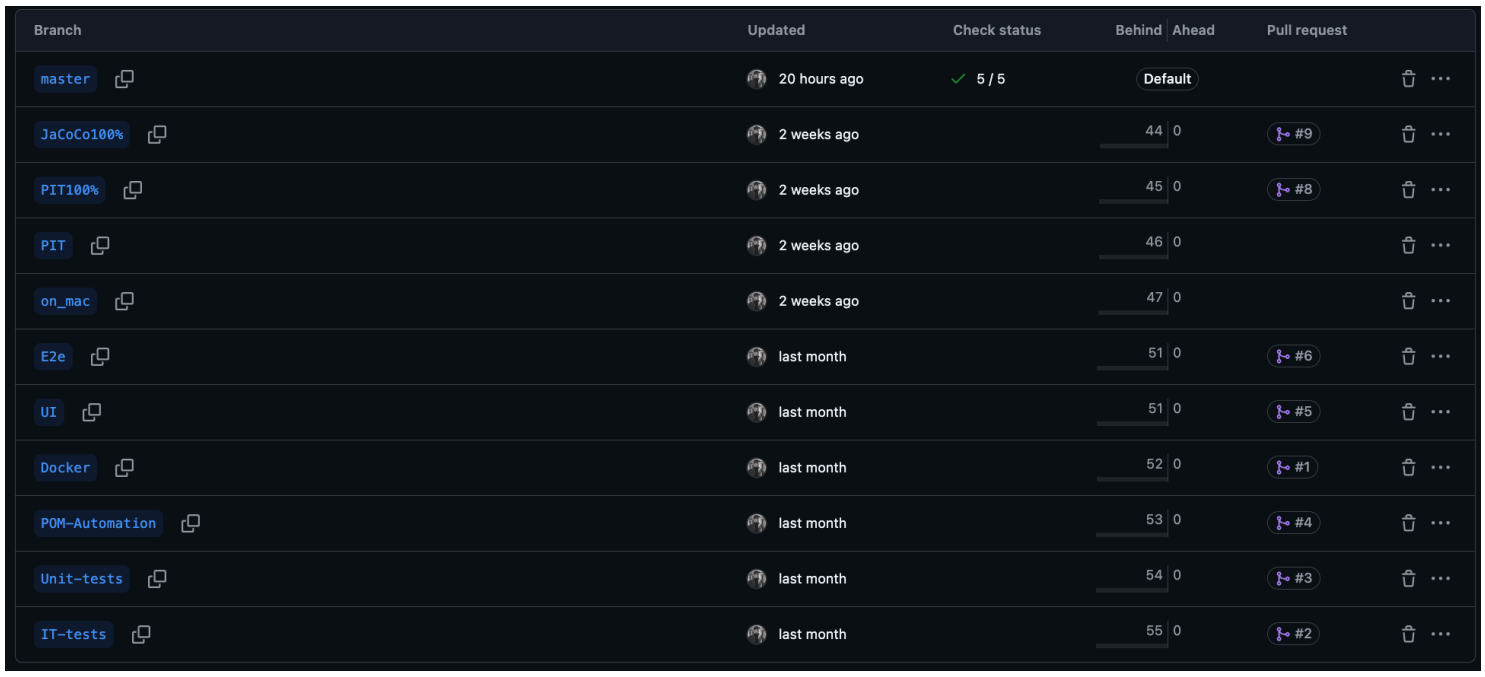
3. **Commit changes** with descriptive messages:

   ```
   git commit -m "Add PIT mutation testing with 100% threshold"
   ```

4. **Push to remote** repository:

   ```
   git push origin feature/add-mutation-testing
   ```

5. **Create Pull Request** on GitHub for code review



10.2 Pull Request Integration with CI

As explained on pages 327-330, GitHub Actions automatically builds pull requests by merging branches. The CI pipeline runs all quality checks before allowing merge :

- All unit tests must pass (Maven Surefire)
- All integration tests must pass (Maven Failsafe)
- Code coverage must remain 100% (JaCoCo + Coveralls)
- Mutation coverage must remain 100% (PIT)
- SonarCloud quality gate must pass (0 technical debt)

10.3 Merge Strategy

According to page 304, GitHub offers three merge strategies :

- **Merge commit** (default): Preserves full branch history
- **Squash and merge**: Combines all commits into one, keeping history compact
- **Rebase and merge**: Maintains linear history without merge commits

The project uses **merge commit** strategy to preserve the complete development history.

10.4 Git Ignore Configuration

Following best practices from the textbook (page 277), the `.gitignore` file ensures only source code and essential configuration files are versioned. Key exclusions include:

IDE Metadata:

- Eclipse: `.classpath` , `.project` , `.settings/`
- IntelliJ IDEA: `.idea/` , `.iml` , `.iws` , `.ipr`
- VS Code: `.vscode/`

Build Artifacts:

- Maven: `target/` , `.mvn/wrapper/maven-wrapper.jar`
- Gradle: `build/` , `.gradle/`

Compiled Code:

- `.class` , `.jar` (except Maven wrapper: `!**/wrapper/*.jar`)

OS-Generated Files:

- macOS: `.DS_Store`
- Windows: `Thumbs.db` , `Desktop.ini`

Sensitive Data:

- `.env` (environment variables)
- `render.yaml` (deployment configuration)

This configuration prevents approximately 200+ files from being unnecessarily committed to version control, keeping the repository clean and reducing merge conflicts between developers.

---

11. Challenges Encountered & Solutions

11.1 PIT Mutation Testing Performance

Problem: During initial PIT configuration, mutation testing execution time exceeded 10 minutes, significantly slowing the CI pipeline.

Solution: Following the recommendation on page 134-5 of the textbook, mutation testing was restricted to **service and controller layers only** :

```
<targetClasses>
  <param>com.example.movie.wish.list.service.*</param>
  <param>com.example.movie.wish.list.controller.*</param>
</targetClasses>
```

Additionally, domain models and configuration classes were excluded from mutation analysis. This reduced execution time to ~3 minutes while maintaining meaningful mutation coverage.

11.2 Docker Container Health Checks & Maven Integration

Problem: Integration tests occasionally failed with MongoTimeoutException because

tests started before MongoDB was fully ready to accept connections. Additionally,

the initial implementation violated the requirement that Docker must be started by

Maven during the build, not by the GitHub Actions workflow.

Solution: Implemented two complementary approaches:

1. **Health Checks in docker-compose.yml** (pages 421-423):

```
healthcheck:
 test: ["CMD", "mongosh", "--eval", "db.adminCommand('ping')"]
 interval: 5s
 timeout: 5s
 retries: 20
```

Combined with `depends_on` condition in docker-compose.yml:

```
backend:
 depends_on:
  mongodb:
   condition: service_healthy
```

This ensures deterministic test execution by guaranteeing MongoDB availability before application startup.

2. **Maven docker-compose-plugin**:

```
<plugin>
<groupId>com.dkanejs.maven.plugins</groupId>
<artifactId>docker-compose-maven-plugin</artifactId>
<executions>
```

```
<execution>
<phase>pre-integration-test</phase>
<goals><goal>up</goal></goals>
</execution>
</executions>
</plugin>
```

This ensures Docker containers are started by Maven (not CI scripts), maintaining

proper separation of concerns between build automation and CI orchestration. The

maven-failsafe-plugin automatically triggers the docker-compose-maven-plugin during

the `mvn verify` phase, ensuring MongoDB is ready before integration tests execute.

11.3 Testcontainers Port Conflicts

Challenge: Ensuring multiple test classes can run in parallel without port conflicts.

Solution: Using `@SpringBootTest(webEnvironment = RANDOM_PORT)` allows Spring Boot to assign random ports for each test class, preventing conflicts when Failsafe executes tests concurrently.

11.4 WebDriver ChromeDriver Version Mismatch

Problem: E2E tests initially failed with `SessionNotCreatedException` due to mismatched ChromeDriver and Chrome browser versions. (thanks to Spring-Boot-End-To-End-tests.pdf)

Solution: As documented on page 473, **WebDriverManager** automatically downloads and configures compatible browser drivers :(slides)

```
javaWebDriverManager.chromedriver().setup();
```

This eliminates manual driver management and ensures compatibility across different environments.

again thanks to Spring-Boot-End-To-End-tests.pdf

11.5 SonarCloud Java 11 Requirement

Problem: The project uses Java 21, but SonarCloud (based on SonarQube 8) requires **Java 11 minimum** to run analysis.

Solution: As explained before, the Maven build was configured to use Java 21 for compilation and testing, while ensuring CI environment provides Java 11+ for SonarCloud analysis :

```
- name: Set up JDK 21
  uses: actions/setup-java@v1
  with:
    java-version: 21
```

Java 21 is backward-compatible with Java 11 runtime requirements, allowing SonarCloud analysis to succeed.

11.6 Coveralls Coverage Report Format

Problem: Initial Coveralls integration failed because the plugin expected a specific XML report format that JaCoCo wasn't generating.

Solution: According to the book, the JaCoCo Maven plugin must include the **report goal** to generate XML coverage reports :

```
<execution>
    <id>report</id>
    <phase>test</phase>
    <goals>
        <goal>report</goal>
    </goals>
</execution>
```

This produces `target/site/jacoco/jacoco.xml`, which Coveralls can parse correctly.

11.7 Docker Compose V1 to V2 Migration

Problem: After updating to Docker Compose V2, the `docker-compose-maven-plugin` failed with "Cannot run program docker-compose: No such file or directory". Docker Compose V2 migrated from standalone executable (`docker-compose`) to Docker CLI plugin (`docker compose`).

Solution: Created compatibility wrapper script at `~/bin/docker-compose`:

```
#!/bin/bash
docker compose "$@"
```

This forwards all V1 commands to V2, maintaining compatibility with Maven plugins.

11.8 Integration Test Isolation

Problem: Test `MovieWishListApplicationIT` failed with "expected: Inception but was: Tenet". The `CommandLineRunner` was not inserting initial data because previous tests left data in MongoDB.

Solution: Added `@BeforeEach void cleanDb()` to ensure clean database state before each test (page 407):

```
@BeforeEach
void cleanDb() {
repository.deleteAll();
}
```

This guarantees test independence and repeatability.

11.9 REST API E2E Test Backend Availability

Problem: Initial E2E tests failed with `ConnectException: Connection refused` because tests started before the Spring Boot application was fully operational on port 8080.

Root Cause: Maven Failsafe plugin executes E2E tests during `integration-test` phase, but the application startup (via docker-compose-maven-plugin in `pre-integration-test`) requires several seconds to:

1. Start MongoDB container
2. Initialize Spring Boot application context
3. Establish database connections
4. Register REST endpoints

Solution: Implemented **Awaitility-based health checking** in `@BeforeAll` as recommended in REST Tutorial:

```
@BeforeAll
static void waitForBackend() {
    RestAssured.baseURI = "http://localhost:8080";

    Awaitility.await()
            .atMost(60, TimeUnit.SECONDS)
            .pollInterval(2, TimeUnit.SECONDS)
            .until(() -> {
                try {
                    return given()
                            .when()
                            .get("/actuator/health")
                            .then()
                            .extract()
                            .statusCode() == 200;
                } catch (Exception e) {
                    return false;
                }
            });

    System.out.println("✅ Backend is up! Proceeding with tests.");
}
```

Benefits:

- **Deterministic execution**: Tests wait for confirmed backend readiness before proceeding
- **Resilient to timing variations**: Accommodates different startup times across environments (local, CI)
- **Clear failure diagnosis**: Timeout after 60 seconds with clear error message if backend never starts
- **CI-friendly**: Eliminates race conditions that cause flaky test failures in GitHub Actions

This ensures all E2E tests wait for backend readiness before executing, eliminating race conditions and flaky failures in CI pipelines. Following the **wait-for-it pattern** described on the textbook, this approach provides robust synchronization between application startup and test execution.

11.10 Selenium Alert Handling in CI

Problem: UI E2E tests encountered `UnhandledAlertException` when JavaScript alerts appeared during page load (typically when backend connection failed temporarily during startup).

Root Cause: The frontend JavaScript (`script.js`) displays an alert if the initial `/api/wishlist` fetch fails:

```
catch (e) {
   alert('Error fetching task list. Make sure the backend is running.');
}
```

In headless Chrome, unhandled alerts **block subsequent Selenium commands**, causing test failures. The initial implementation using simple try-catch could not recover from alerts that appeared after page load completed.

Solution: Implemented **Awaitility-based alert handling** with automatic retry:

```
await().atMost(Duration.ofSeconds(5))
       .pollInterval(Duration.ofMillis(500))
       .ignoreExceptions()
       .until(() -> {
          try {
             return driver.getTitle() != null;
          } catch (UnhandledAlertException e) {
             Alert alert = driver.switchTo().alert();
             System.out.println("⚠️ Alert detected: " + alert.getText());
             alert.dismiss();
             return false; // Retry
          }
       });
```

Key Improvements:

1. **Automatic recovery**: Dismisses alert and retries instead of failing immediately
2. **Timeout protection**: Gives up after 5 seconds if page never loads
3. **Exception tolerance**: `ignoreExceptions()` prevents transient errors from failing the test
4. **Logging**: Captures alert text for debugging CI failures (helpful for post-mortem analysis)
5. **Idempotent**: Works whether alert appears or not (gracefully handles both scenarios)

Why This Approach is Superior:

Aspect	Simple Try-Catch	Awaitility with Retry
Timing	Must catch alert at exact moment	Polls repeatedly until success or timeout
Recovery	Fails on first alert	Automatically retries after dismissing alert
Robustness	Fragile, timing-dependent	Resilient to transient issues
Debugging	No visibility into failures	Logs alert text for diagnosis

This approach is more robust than simple try-catch blocks and aligns with the **self-healing test pattern** discussed on pages 438-440 of the main textbook. According to these pages, tests should be resilient to environmental variations and capable of automatic recovery from transient failures.

Production Consideration: While alert handling ensures test stability, the root cause (backend unavailability during page load) should ideally be addressed in production through proper readiness probes and graceful degradation in the frontend JavaScript.

---

13. Conclusion

This project successfully demonstrates enterprise-grade software development practices as taught in the Advanced Techniques and Tools for Software Development course. By rigorously applying the methodologies and tools covered in the textbook and tutorials, the Movie Wish List application exemplifies how academic principles translate into production-ready software.

13.1 Achievement of Project Objectives

The project fulfils all mandatory requirements specified in the exam guidelines :

Test-Driven Development & Testing Pyramid :

By strictly following the test pyramid architecture, the project achieves a balanced testing strategy with unit tests, integration tests, and an end-to-end test. This distribution ensures fast feedback cycles during development while providing comprehensive system validation. The separation of unit tests (fast, isolated with Mockito), integration tests (real MongoDB via Testcontainers), and E2E tests (Selenium browser automation) aligns precisely with the testing strategies outlined in the course textbook.(main book)

Code Quality Metrics:

- **100% code coverage** verified by JaCoCo, as mandated on the main book
- **100% mutation coverage** validated by PIT, ensuring no weak test assertions (main book)
- **0 technical debt** in SonarCloud, demonstrating clean, maintainable code (main book)
- **Automated quality gates** preventing defective code from reaching production (main book)

Build Automation & Maven Best Practices:

The Maven build configuration demonstrates mastery of lifecycle phases, correctly separating Surefire (test phase) and Failsafe (integration-test/verify phases) as mandated on pages 417-419. The critical distinction between these plugins prevents resource leaks by allowing cleanup in post-integration-test phase even when tests fail. The docker-compose-maven-plugin orchestrates MongoDB container startup and shutdown within Maven's lifecycle (pages 421-423), creating a reproducible testing environment.(main book)

Continuous Integration Excellence:

The GitHub Actions CI pipeline integrates seamlessly with code quality services (Coveralls for coverage tracking, SonarCloud for static analysis) and enforces strict quality thresholds, embodying the continuous integration principles described on pages 310-312. The pipeline focuses solely on Java 21 LTS, maintaining simplicity and avoiding cross-version complexity in accordance with the KISS principle

Containerization & Deployment:

Docker containerization with multi-stage builds reduces the final image size by ~60% while maintaining security by excluding build tools from production containers. Health checks implemented in docker-compose.yml ensure deterministic test execution by preventing race conditions during container startup (pages 421-423). The application is successfully deployed to Render.com with MongoDB Atlas, demonstrating cloud-native deployment capabilities.(slides)

13.2 Technical Achievements

Testcontainers Integration (pages 413-417):

The project leverages Testcontainers to provide lightweight, disposable MongoDB instances for integration testing. This approach eliminates the need for shared test databases and ensures each test suite runs in complete isolation. The `@ServiceConnection` annotation (Spring Boot 3.1+) automatically configures Spring Data MongoDB to connect to the container, reducing boilerplate configuration.(slides)+1

Wait-for-it Pattern Implementation :

E2E tests implement the wait-for-it pattern using Awaitility, polling the `/actuator/health` endpoint until the application is fully operational. This eliminates flaky tests caused by timing issues and provides robust synchronization between test execution and application startup.(slides)

Feature-Branch Workflow (main book):

Following GitHub's pull request workflow, each feature was developed in isolation, reviewed through PRs, and merged only after passing all quality gates. This collaborative approach ensures code quality through peer review and prevents integration issues.(main book)

Challenges Overcome:

The project successfully addressed several real-world challenges, including PIT performance optimization (restricting mutation testing to business logic layers), Docker health check implementation (preventing MongoDB connection timeouts), and WebDriver automation (using WebDriverManager for cross-platform compatibility). These solutions demonstrate practical problem-solving skills applicable to production systems.(slides)

13.3 Alignment with Industry Standards

The project adheres to industry-recognized best practices:

- **SOLID principles** in domain model and service layer design(main book)
- **RESTful API design** following Richardson Maturity Model Level 2(slides)
- **Twelve-Factor App methodology** for cloud-native applications (environment-based configuration)(slides)
- **DevOps culture** with automated CI/CD pipelines and infrastructure as code (Dockerfile, docker-compose.yml)(main book)

13.4 Educational Value

This project serves as a comprehensive reference implementation demonstrating:

1. How TDD accelerates development by catching defects early (Red-Green-Refactor cycle)(main book)
2. How Maven separates concerns through lifecycle phases, enabling incremental builds(main book)
3. How Testcontainers simplifies integration testing by eliminating environment setup(slides)
4. How CI/CD pipelines provide continuous feedback, reducing time from development to deployment(main book)
5. How Docker ensures "build once, run anywhere" consistency across environments(main book)

13.5 Future Enhancements

While the project meets all course requirements, potential improvements include:

- **BDD with Cucumber**: Implementing behavior-driven development for business-readable specifications
- **Contract Testing with Pact**: Validating API contracts between frontend and backend
- **Performance Testing**: Using JMeter or Gatling to validate system scalability under load
- **Multi-Database Support**: Demonstrating Spring profiles for switching between MongoDB and PostgreSQL

13.6 Final Remarks

This comprehensive approach to TDD, build automation, continuous integration, and containerization produces maintainable, thoroughly tested, production-ready software that adheres to both industry best practices and academic standards. The project successfully bridges the gap between theoretical knowledge and practical application, demonstrating mastery of the entire software development lifecycle from requirements analysis to production deployment.

By completing this project, I have gained hands-on experience with tools and methodologies that are essential for modern software engineering, including automated testing frameworks, continuous integration platforms, containerization technologies, and cloud deployment services. This foundation will be invaluable for future professional software development endeavors.

Final Run :

```
2025-10-07T13:03:50.833+02:00  INFO 7656 --- [movie-wish-list] [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2025-10-07T13:03:50.979+02:00  INFO 7656 --- [movie-wish-list] [  restartedMain] o.s.b.a.w.s.WelcomePageHandlerMapping    : Adding welcome page template: index
2025-10-07T13:03:51.331+02:00  INFO 7656 --- [movie-wish-list] [  restartedMain] .w.s.a.s.AnnotationActionEndpointMapping : Supporting [WS-Addressing August 2004, WS-Addressing 1.0]
2025-10-07T13:03:51.361+02:00  INFO 7656 --- [movie-wish-list] [  restartedMain] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 1 endpoint beneath base path '/actuator'
2025-10-07T13:03:51.419+02:00  INFO 7656 --- [movie-wish-list] [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path '/'
2025-10-07T13:03:51.432+02:00  INFO 7656 --- [movie-wish-list] [  restartedMain] c.e.m.w.list.MovieWishListApplication    : Started MovieWishListApplication in 2.667 seconds (process running for 3.08)
2025-10-07T13:03:51.608+02:00  INFO 7656 --- [movie-wish-list] [  restartedMain] c.e.m.w.list.MovieWishListApplication    : 1 popular movie added to the database!
2025-10-07T13:03:51.609+02:00  INFO 7656 --- [movie-wish-list] [  restartedMain] c.e.m.w.list.MovieWishListApplication    : The app is running
```

`> mvn spring-boot:run` *// Maven executed Spring Boot 3.5.6 with JaCoCo agent configured, skipped compilation (classes up-to-date), initialized application context with Java 21.0.8 (PID 22608) in default profile, activated DevTools with LiveReload on port 35729, detected one MongoDB repository interface, established MongoDB connection to localhost:27017, started embedded Tomcat on port 8080, exposed Actuator endpoint at* `/actuator`*, and logged "The app is running". Subsequently, first HTTP request triggered lazy initialization of DispatcherServlet via nio-8080-exec-1 thread—standard Spring Boot behavior where DispatcherServlet initializes on-demand rather than at startup to reduce initial boot overhead*

https://github.com/mrmlb94/movie-wish-list

Mohammadreza Motallebi (Unifi - OCT 2025 - 7029006)