Testing the Task Manager — Assignment

📋 **What You'll Build** You will write two types of tests for the Task Manager application: **unit tests** for the service layer and **integration tests** for the controller layer. By the end, you'll understand the fundamental difference between these two testing strategies and why both matter.

🎯 Your Tasks

1. Create `TaskServiceTest.java` — a unit test for the `TaskService` class
2. Create `TaskControllerIntegrationTest.java` — an integration test for the task controller endpoints

---

🧠 Core Concepts to Understand First

Unit Testing vs Integration Testing

These are not the same thing, and choosing the wrong one for a given scenario will either give you false confidence or make your tests unnecessarily slow.

Unit Testing isolates a single class and tests its logic in complete isolation. Every dependency the class has is replaced with a *fake* (a "mock"). You're answering the question: *"Does this class's logic work correctly, given controlled inputs?"*

Integration Testing wires up the whole (or most of the) Spring application context and tests how components work *together* — HTTP layer, service layer, database, security filters, all of it. You're answering: *"Does this feature work end-to-end as the user would experience it?"*

🔥 **Rule of Thumb** Unit tests are fast (milliseconds) because nothing real runs. Integration tests are slow (seconds) because Spring actually boots up. Write many unit tests and fewer integration tests.

What Is Mocking?

When you unit test `TaskService`, it has real dependencies: `TaskRepository`, `UserRepository`, `TaskMetrics`, and `SecurityContext`. You don't want to spin up a real database for a unit test. Mocking solves this.

A **mock** is a fake object that:

- Implements the same interface as the real thing
- Does nothing by default (returns `null`, `0`, or empty)
- Lets you *program* specific return values with `when(...).thenReturn(...)`
- Lets you *verify* that certain methods were called with `verify(...)`

Think of it like a stunt double in a film — it looks like the real actor from the camera's perspective, but you control exactly what it does.

✏️ **Mockito** The testing framework you'll use is **Mockito**, which is the standard mocking library in the Java ecosystem. Spring Boot includes it automatically in test dependencies.

The `SecurityContext` Problem

`TaskService` calls `SecurityContextHolder.getContext().getAuthentication().getName()` internally to find out which user is making the request. In a real request, the `JwtAuthenticationFilter` populates this. But in a unit test, there's no HTTP request, no filter chain — the `SecurityContextHolder` is empty.

You need to **manually populate** the `SecurityContextHolder` in your test setup so that when `TaskService` asks *"who is the current user?"*, it gets a sensible answer.

How to Approach This

- Mock both `SecurityContext` and `Authentication` as separate mock objects
- Make `securityContext.getAuthentication()` return your mock `Authentication`
- Make `authentication.getName()` return a test username string
- Then set your mock `SecurityContext` into the real `SecurityContextHolder`

This is the most tricky part of the unit test — once this is working, the rest follows naturally.

`@ExtendWith(MockitoExtension.class)`

This annotation tells JUnit 5 to activate Mockito's extension. Without it, `@Mock` and `@InjectMocks` annotations do nothing. It replaces the old JUnit 4 `@RunWith(MockitoJUnitRunner.class)`.

`@Mock` vs `@InjectMocks`

Annotation	Purpose
`@Mock`	Creates a fake (mock) instance of that type
`@InjectMocks`	Creates a **real** instance of the class being tested, and automatically injects all `@Mock` fields into it

So in your test, `TaskService` will be a real object running real code — but every dependency it calls will be a mock you control.

`@BeforeEach`

A method annotated with `@BeforeEach` runs **before every single test method** in the class. Use it to:

- Create test data objects (a test `User`, a test `Task`)
- Set up common mock behaviors that all tests need
- Wire the `SecurityContextHolder` with your mock context

This keeps your individual `@Test` methods clean and focused.

Writing Assertions

JUnit 5's `Assertions` class provides the methods you'll use to verify outcomes:

- `assertNotNull(result)` — confirms something was returned
- `assertEquals(expected, actual)` — confirms a value matches what you expect
- `assertTrue(condition)` — confirms a boolean is true
- `assertThrows(ExceptionClass.class, () → { … })` — confirms code throws a specific exception

Always Assert Something Meaningful A test that only calls the method without asserting anything is not a real test — it only proves the code doesn't throw an exception. Assert the *specific outcome* you care about.

`verify()` — Testing Behavior, Not Just Output

Sometimes the important thing isn't what a method *returns* but what it *does*. For example, when creating a task, you want to confirm that `taskRepository.save()` was actually called and that `taskMetrics.incrementTaskCreated()` was triggered.

`verify(mockObject).methodName(args)` checks that a method was called exactly once. If it wasn't called, the test fails.

Use `any(ClassName.class)` as an argument matcher inside `verify()` when you don't care about the exact object passed, only that the method was called.

Integration Testing Concepts

`@SpringBootTest`

This annotation tells Spring to boot up the **full application context** — all beans, security config, database, everything — just like a real running application. This is what makes integration tests slow but also what makes them realistic.

`@AutoConfigureMockMvc`

Instead of starting a real HTTP server on a port, this configures a `MockMvc` object that simulates HTTP requests inside the test JVM. You get realistic request/response handling without network overhead.

`MockMvc` — How It Works

`MockMvc` lets you craft HTTP requests programmatically and make assertions on the response. The pattern looks like:

```
mockMvc.perform( [HTTP method and URL] )
       .andExpect( [assertion about the response] )
```

You can chain multiple `.andExpect()` calls to check status code, response body fields, headers, and more.

Useful Methods to Know

- `post("/some/url")` / `get("/some/url")` — build a request
- `.contentType(MediaType.APPLICATION_JSON)` — set the Content-Type header
- `.content(jsonString)` — attach a request body

- `.header("Authorization", "Bearer " + token)` — attach headers
- `status().isOk()` / `status().isCreated()` — assert HTTP status
- `jsonPath("$.fieldName").value("expectedValue")` — assert a specific JSON field in the response body

`ObjectMapper` — Java ↔ JSON Conversion

`ObjectMapper` (from Jackson) converts between Java objects and JSON strings in both directions:

- `.writeValueAsString(object)` → JSON string (for request bodies)
- `.readTree(jsonString).get("fieldName").asText()` → extract a field from a JSON response

The Authentication Problem in Integration Tests

Your protected endpoints require a valid JWT in the `Authorization` header. You can't hardcode one — they expire. The solution is to **actually log in as part of test setup**.

In `@BeforeEach`, perform a real login request against `/api/auth/login`, extract the token from the response JSON, and store it. Then use that token in every subsequent test request.

Test Data Dependency This approach means your integration test depends on a user (e.g., `admin` / `admin123`) actually existing in the database. Your `DataInitializer` seeds this user on startup, which is why it runs before the tests can succeed. This is an example of *test fixture* setup.

Where to Put Your Files

```
src/
└── test/
    └── java/
        └── com/nick/taskmanager/
            ├── service/
            │   └── TaskServiceTest.java        ← Unit test
            └── controller/
                └── TaskControllerIntegrationTest.java  ← Integration test
```

⚠ Package Matters The test classes must be in the same package as the classes they test (even though they live under `src/test/`). This gives them access to package-private members and ensures Spring component scanning works correctly for integration tests.

▶️ Running Your Tests

```shell
# Run all tests
mvn test

# Run only one specific test class by name
mvn test -Dtest=TaskServiceTest

# Run with a coverage report (generates HTML report in target/site/jacoco/)
mvn clean test jacoco:report
```

After running with Jacoco, open `target/site/jacoco/index.html` in a browser to see which lines of your code are covered by tests and which are not.

✅ Checklist Before Submitting

- [ ] Unit test class uses `@ExtendWith(MockitoExtension.class)`
- [ ] All dependencies of `TaskService` are declared as `@Mock`
- [ ] `TaskService` itself is declared as `@InjectMocks`
- [ ] `SecurityContextHolder` is populated in `@BeforeEach`
- [ ] Each test has at least one `assert` statement
- [ ] Each test verifies meaningful behavior with `verify()`
- [ ] Integration test class uses both `@SpringBootTest` and `@AutoConfigureMockMvc`
- [ ] JWT token is obtained via a real login call in `@BeforeEach`
- [ ] Every protected endpoint test attaches the `Authorization` header
- [ ] All tests pass with `mvn test`

💡 Common Mistakes to Avoid

⚠ **Don't Mock the Class Under Test** A common beginner mistake is accidentally putting `@Mock` on the class you're *testing* (`TaskService`). That would create a fake service that does nothing. The class being tested gets `@InjectMocks`.

⚠ **Mockito Strict Stubbing** `MockitoExtension` uses strict stubbing by default. If you set up a `when( … ).thenReturn( … )` that no test actually uses, Mockito will fail the test with an "unnecessary stubbing" error. Only stub what each test actually needs — put shared, always-needed stubs in `@BeforeEach` and test-specific ones inside the `@Test` method.

⚠ **Forgetting the `Content-Type` Header** In integration tests, if you send a JSON body without `.contentType(MediaType.APPLICATION_JSON)`, Spring won't know how to deserialize it and will return `415 Unsupported Media Type`. Always set the content type when sending a body.

🔗 Related Notes

- Spring Security with JWT — How the `SecurityContextHolder` gets populated in real requests

- Spring Boot Auto-configuration — What `@SpringBootTest` actually boots up
- Mockito Documentation — Full reference for mocking patterns