Chapter 5. Automated Testing

> Quality is not an act, it is a habit.
>
> Philosopher Will Durant, paraphrasing Aristotle

When a new year begins, many set goals to live and maintain a healthier lifestyle. Those goals aren't achieved by eating a single healthy meal or by going to the gym one time. It is about *consistently* making healthy choices most of the time. Similarly, producing high-quality, maintainable software is the result of *consistently* practicing good habits when it comes to writing, reviewing, and testing your code. Like maintaining a healthy lifestyle, this doesn't come easy, and excuses only set you back.

However, the path to consistency isn't always smooth. Many developers struggle with self-doubt, wondering if they're making the right decisions or if their problem-solving approach will be questioned. Writing high-quality software is a discipline that requires regular practice, but over time, your testing efforts will serve as effective safeguards that your teammates and future self will appreciate. In this chapter, you will learn the benefits of automated testing, the different types of tests you will encounter, and how to write them.

Benefits of Automated Testing

It's natural to be skeptical of writing additional code to verify your existing code, and it can sometimes feel not worth the effort. You might wonder if it's just another trend or resume-building exercise. However, automated testing is far more than that. It's a valuable skill to have and an important investment in your codebase.

In the following sections, you'll explore the concrete benefits of writing automated tests. You'll examine how this practice can enhance your code quality, boost your confidence in your work, and ultimately make you a more efficient engineer. By understanding these advantages, you'll gain a clear focus for your testing efforts and appreciate why investing time in automated testing is worthwhile for your professional growth.

Acts as Documentation

Joining an existing project can be overwhelming (see Chapter 6 for what to do when joining an existing project). It's a luxury for a project to have documentation.[1] If you're fortunate, you might be able to confer with lead developers or domain experts to get a high-level overview before you begin work. This isn't always possible, and documentation can be sparse or even nonexistent. Your saving grace is a project that has well-written tests.

Consider a scenario where you're tasked with fixing payment-processing issues with a particular type of card. Without proper documentation or tests, you'd have

to sift through unfamiliar (likely complex) code or rely on colleagues for guidance.

Now imagine finding a comprehensive test suite for payments. You'd see a `PaymentProcesserTests` class with descriptive test names like these:

- `shouldValidateValidCreditCardNumber`
- `shouldProcessCreditCardTransaction`
- `shouldProcessOrangePay`

These tests provide insight into the module's features and point to relevant code sections. Modern IDEs allow easy navigation to the specific services being tested (see Chapter 2 for effective code-reading strategies). A failing test leads you to the likely culprit within the codebase.

You might discover a test named `shouldFailWhenCreditCardTypeIsOrangePay`, revealing that Orange Pay isn't supported. This information helps you quickly identify and fix the issue, update the frontend, and improve documentation.

This example demonstrates how tests can serve as documentation, helping you learn a codebase faster. By focusing on improving test suites, you can enhance code quality, reduce production issues, and speed up your ability to contribute to projects.

Improves Maintainability

Writing good, maintainable code is a skill that typically takes years to develop (see Chapter 3 for more on this). Writing tests first is like thinking before you speak: it helps you plan and structure your thoughts. The primary goal of beginners is often just to make code work. While this remains important, writing tests help you recognize areas for improvement much sooner.

Consider this example of a `BlogPostController` (code has been omitted for brevity):

```
public class BlogPostController {
   public void publish(Post post) {
       // save blog post to the database (flip isPublished to true)
       // log blog post has been published
       // send email to subscribers about the new post
   }
}
```

At first glance, this might seem perfectly acceptable. However, if you attempt to write tests for this method, you'll soon realize it's doing too much:

- Communicating with a database
- Logging information
- Sending emails

This approach violates the single responsibility principle, which states: "A class should have only one reason to change."

By thinking about how to test this code, you naturally identify its flaws. You can then refactor each part of the publishing process into separate classes, making them easier to test and maintain.

Writing tests not only verifies functionality but also guides you toward better code design. It helps you spot potential issues before they become problems, leading to more maintainable and reusable software.

Boosts Your Confidence

Among the many benefits of automated testing, one stands out: the confidence to code freely. As in many scenarios in life, projecting confidence can help you deal with pressure and tackle personal and professional challenges. Software development is inherently iterative, as we constantly write, experiment, and refactor. Without tests, you might code with caution, trying to avoid introducing changes that break things. You might be afraid to try new techniques or a creative solution. But with a robust test suite, you are given a safety net. This freedom fundamentally changes how you approach coding. You can make changes, try new solutions, and refactor with confidence knowing your tests will catch any issues.

Changing and refactoring code can be a stressful experience. You might not have a complete understanding of the application, and side effects can be difficult to predict. Working without tests is like climbing a mountain without ropes. If everything goes well, it can be an adrenaline rush, but one mistake can be catastrophic.

Tests act as a safety net when navigating a codebase and give you the ability to act with confidence. While your initial goal is to make a feature work, you need to consider how your changes might affect the entire system. Without comprehensive tests, everything might appear functional and then result in issues during production.

A robust test suite allows the team to do the following:

- Refactor code confidently
- Ensure that changes don’t break existing functionality
- Identify potential issues before they reach production

By writing and maintaining a comprehensive set of tests, we can do the following:

- Better understand the system’s behavior
- Catch bugs early in the development process when they are simpler and cheaper to fix
- Reduce the likelihood of introducing regressions
- Ensure features meet specifications and maintain code quality through automated testing

This approach not only improves code quality but also boosts your confidence as a developer. You can make changes, add features, and refactor with the assurance that your tests will catch potential issues.

Confidence in coding comes from a combination of knowledge, experience, and tools. Automated tests are a powerful partner in building and maintaining confidence throughout your development career.

Leads to Consistency and Repeatability

At one time, developers often relied on countless hours of manual testing. You'd create a list of steps to put stress on your new feature. While you might have followed the script most of the time, a manual approach is prone to errors, isn't repeatable, and is very labor and time intensive. Human beings are inherently incapable of performing the same task repeatedly without variation.

Automated testing delivers consistent, repeatable results. Unlike humans, test scripts follow the exact same steps every time without errors or omissions. These tests run quickly with minimal effort at the click of a button or during builds. Regression tests specifically verify that existing functionality continues to work as your code changes.

With a comprehensive suite of tests, you can ensure they run consistently and repeatedly, regardless of the environment. Automated tests can be executed frequently,[2] providing effective regression testing. These tests ensure that previously fixed bugs don't mysteriously return; the tests also identify new bugs introduced by changes in the codebase. When testing a new feature, skip the expensive manual testing and rely on automated tests.

Types of Automated Testing

Now that you know why you should test, let's explore what you should (and should not!) test. *Automated testing* is a broad category that covers UI tests, end-to-end tests, and integration tests, as well as unit tests. It is common to think of these types of testing as a pyramid, shown in Figure 5-1.

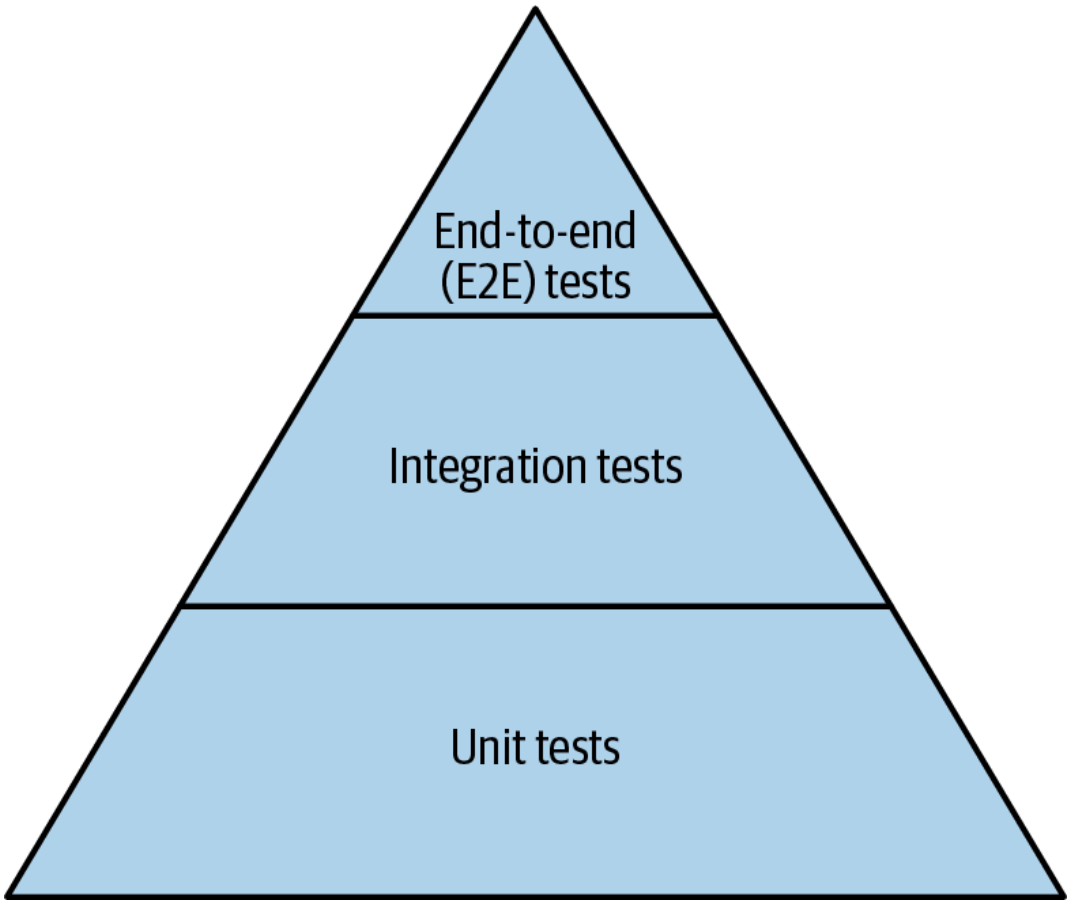

Figure 5-1. Pyramid showing the three types of automated tests you will have to write, and the recommended amount of each type relative to the others

The *testing pyramid* is an important concept in software testing, yet it's often misunderstood or ignored entirely. Created by Mike Cohn, this model provides visual guidance for the types of tests you should have in your application.

These are the three types of automated testing:

Unit tests
: Designed to cover individual components for functions in isolation from the rest of the system, ensuring that each part works correctly on its own

Integration tests
: Verify how the different components or modules of a system work together as a cohesive unit

End-to-end tests
: Cover the entire application, starting from the user interface and extending all the way to the backend system

This structure is based on fundamental trade-offs in software testing. As you move up the pyramid, tests become slower to run, more expensive to maintain, and more likely to break. A unit test might run in microseconds, while an end-to-end test could take several minutes. When you have hundreds or thousands of tests, these differences add up dramatically. Let's take a closer look at each one.

Unit Tests

Unit tests form the foundation of the testing pyramid, representing the largest portion of your test suite. Think of them as a contract that your code must fulfill. If the code changes in any functional way, the corresponding unit tests should break, alerting you to potential issues. These tests examine individual components or functions in isolation, ensuring that each piece works correctly on its own.

Unit tests serve as your first line of defense against defects, working alongside static analysis and code reviews. They should be quick to write and execute, providing rapid feedback during development. When a unit test fails, it typically points to a specific function or line of code, making debugging straightforward. This speed and precision make unit tests invaluable during feature development and continuous integration, giving developers the confidence to iterate and improve their code quickly.

Integration Tests

An *integration test* is a detailed process that thoroughly verifies how different components or modules of a system work together as a cohesive unit. While unit tests focus on individual parts of the code in isolation, integration tests cover a broader scope by examining the interactions between these parts.

However, they are still narrower in focus compared to full system tests, which evaluate the entire system's performance and functionality. Integration tests are crucial because they help catch issues that arise only when individually tested components are combined, ensuring that the integrated system functions correctly and efficiently. This step is essential in the SDLC to maintain the integrity and reliability of the system as a whole.

Because of their scope and complexity, integration tests typically take longer to run than unit tests and are often executed less frequently. When integration tests fail, more investigation is usually needed to identify which interaction between modules caused the issue.

End-to-End Tests

An *end-to-end (E2E) test* is a comprehensive testing procedure that covers the entire application, starting from the user interface and extending all the way to the backend systems. These tests are designed to simulate real user scenarios, ensuring that the application functions as expected in a real-world environment.

Because of their complex and thorough nature, these tests are typically slow to execute and require significant resources and time to run. As a result, you will generally find fewer end-to-end tests compared to other types. They are often run less frequently, usually as part of a scheduled process. End-to-end tests can be brittle and are more susceptible to false negatives (it is possible for a test to fail because of a trivial change to a user interface element as opposed to an actual issue with the functionality). Failing end-to-end tests may require substantial debugging efforts to identify and rectify the issue.

Despite this, end-to-end tests are invaluable as they provide a high amount of confidence in the system's overall functionality, helping to ensure that all components of the application work together seamlessly.

What Mix of Tests Should You Be Writing?

The actual number of tests will vary from project to project, depending on the complexity of the application, the technology stack being used, and the criticality of the application. Additionally, the number of tests could be based on your organization's standards and practices, which may include specific guidelines or benchmarks for test coverage and quality. It's important to tailor your testing strategy to suit the unique needs of your project while ensuring that you maintain a balanced approach covering all necessary aspects of the application.

What You Should Not Test

While comprehensive testing is essential for software quality, it's equally important to be strategic about what you test. Here are key guidelines for what to avoid in your testing strategy:

- Your focus should be on your code, and you should not be testing language features or framework code.[3]
- Avoid testing generated code like getters, setters, builder methods, and auto-generated data transfer objects (DTOs).
- Avoid testing private methods directly. Instead, focus on the public interface that calls these private methods.
- Avoid writing tests that depend on external services. Instead, use mocks for unit tests or test doubles for integration tests.

A well-designed test suite should be comprehensive yet maintainable, providing thorough coverage without becoming unwieldy. By focusing on testing your own code and avoiding unnecessary tests, you can create more effective and efficient test suites. Now that you understand the types of automated tests that exist and where you should and should not focus your efforts, it's time to talk about code coverage.

Code Coverage

Code coverage is a metric used to measure the percentage of your code that is executed when your tests are run. Think of code coverage like a map in a video game. When you begin, the entire map is dark, but as you begin to move around and explore areas, they become visible. Tests that execute parts of your code “light up” those sections, showing you’ve been there. High coverage means you explored most of the map, while low coverage means there are blind spots where bugs can be hiding.

Because code coverage is important in the world of testing, you will find a variety of IDEs that support code coverage as well as tools for the language of your choice. As a developer, you can run coverage right in your IDE to get instant feedback on your coverage. There are also tools that integrate into your CI/CD pipeline to enforce a minimum code coverage threshold. If the coverage falls below that, they can fail, which in turn blocks a PR from being merged.

Code coverage tools analyze your test runs and generate reports showing exactly which lines, branches, and functions were executed. You might see percentages like “85% line coverage” or “72% branch coverage,” indicating how much of your code was executed during testing.

While code coverage provides valuable insights, it’s important to understand its limitations. A common misconception is that higher code coverage automatically means better testing. Some organizations will even require minimum coverage thresholds of 80%–90% coverage before it can be deployed. While there is good intention behind these requirements, they can also lead to counterproductive behaviors.

As a developer, you can write tests that execute code without actually verifying meaningful behavior, essentially “gaming” the coverage metrics. A test that calls a method but doesn’t assert anything useful still counts toward coverage yet provides no real protection against bugs.

Instead of striving for 100% coverage, which is often a vanity metric, use it as a feedback mechanism for the tests you’re writing. Low coverage in mission-critical business logic might indicate an area of the code that needs more attention. High coverage in areas where you’re simply testing data carrier classes or configuration might suggest that you’re testing code that doesn’t require it.

Focus on writing meaningful tests that verify behavior and let code coverage guide you to areas of your code needing more attention. The goal isn’t to hit an arbitrary number, just to meet a requirement. The point of testing is to build confidence in the code you’re writing and ultimately shipping to production. Now that you understand code coverage, let’s start writing some tests so you can begin using these tools.

Writing Tests

It’s time to get down to business and learn the mechanics of writing tests.

Getting Started

No matter what programming language, framework, or meta-framework you’re using, multiple testing tools are likely available to you. In this section, we are

going to use the Java programming language. While the setup and syntax might be different for your language of choice, the ideas remain the same.

Once you have created a project, your next decision will be the testing framework to use. There are many great options to choose from, but the most popular testing framework for Java is called JUnit. To include JUnit, you can declare the appropriate dependencies in your Maven POM file and install them.

We won't cover the details of JUnit here, but if you're interested in learning more about it, you should check out the excellent documentation in its user guide. Once you have JUnit installed, you can begin writing tests.

There are two main approaches to writing tests, test first and test last:

- *Test-first methodologies*, such as test-driven development (TDD), follow a "red-green-refactor" cycle. First, you write a failing test ("red"), then implement the minimum code to pass the test ("green"), and finally improve the code without changing its behavior ("refactor").
- *Test-last approaches* involve writing tests after implementing the functionality.

Both methods have their merits, and deciding between them often depends on project requirements and team preferences. Regardless of the approach, all testing methodologies use assertions to verify expected outcomes.

AI Note

Writing tests is an excellent use case for AI assistance. You can provide AI tools with examples from your existing test suite to help them understand your organization's preferred testing style, naming conventions, and coding standards. AI can help generate test cases, suggest edge cases you might have missed, write boilerplate test code, and even help refactor existing tests for better readability. This can significantly speed up the testing process while maintaining consistency with your team's established patterns.

Assertions

An *assertion* is a function or method that can be used to verify that certain conditions are met during the execution of a program. There are a number of assertion libraries in Java, all offering different features.

JUnit 5 comes with a number of assertions for performing a wide range of verifications. All of the assertions are static methods in the `org.junit.jupiter.Assertions` class. Let's say you want to verify that the sum of a mathematical operation is correct. There is an `assertEquals` method that will take two integers, shown in the following code block. The first argument is the expected value, and the second argument is the actual value:

```
@Test
void shouldAdd2Numbers() {
   int expected = 3;
   int actual = 2 + 1;
   assertEquals(expected,actual);
}
```

If the two numbers are equal, the assertion passes and therefore so does this test. If they are not equal, the assertion fails and causes the test to fail. To get familiar

with available assertions, read through the documentation for whatever assertion library you are using.

Writing Unit Tests

Remember, unit tests are isolated tests that run independently and very fast. Imagine a class that performs mathematical operations necessitating tests for each of them. Here is an example class called `Operations`:

```
package com.fose;

public class Operations {
   public double add(double a, double b) {
       return a + b;
   }
   public double subtract(double a, double b) {
       return a - b;
   }
}
```

Before you can create your first test, you need to pay attention to where your source code is located. Code for this class is in the package `com.fose`. When you create a test class, you will want to place it in */src/test/java* and in the same package that your source code is in. In the following example, we have an add and subtract test to cover all of the functionality from the `Operations` class.

Here you are following test-last approaches, which involve writing tests after implementing the functionality. You can start by adding the method stubs for each of the tests you're going to write:

```
package com.fose;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class OperationsTest {
   @Test
   void add() {
   }

   @Test
   void subtract() {
   }
}
```

First, you need an instance of the `Operations` class. You will often hear this referred to as the *system under test* (*SUT*) because it refers to the specific component, module, or part of your system that you are currently testing. This is a unit test that is focused on testing the `Operations` class and nothing else:

```
class OperationsTest {

   private final Operations sut = new Operations();
}
```

With an instance of the system under test available, you can now fill in the test methods. We aren't doing anything special here, but you will want to make sure you cover any edge cases because you don't want your users to catch them for you:

```
package com.fose;
```

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class OperationsTest {
   private final Operations sut = new Operations();
   @Test
   void add() {
       assertEquals(5, sut.add(2,3));
       assertEquals(0, sut.add(-1,1));
       assertEquals(-5, sut.add(-2,-3));
   }
   // Additional tests omitted for brevity
}
```

Mocking

In software testing, *mocking* is a technique used to create a simulated version of a dependency (often called a *stunt double* in this context) that allows you to test code in isolation.

What is a dependency? In the context of writing a unit test, a *dependency* is any other class the system under test relies on to complete a task. While mocking helps you isolate dependencies in unit tests, it presents a significant challenge in integration testing. In integration tests, you aim to verify how these dependencies work together, which can be tricky. Managing multiple dependencies, ensuring they're in the correct state, and handling their interactions can quickly complicate your integration tests. It's like trying to juggle multiple balls at once—you need to keep track of how each dependency behaves and impacts the others.

The following example has a class called `UserService` with two dependencies: `UserRepository` and `EmailService`. The `UserRepository` class is responsible for reading and persisting users to your database while `EmailService` is responsible for sending out emails:

```
package com.fose;

public class UserService {

   private UserRepository userRepository;
   private EmailService emailService;

   public UserService(UserRepository userRepository, EmailService emailService) {
       this.userRepository = userRepository;
       this.emailService = emailService;
   }

   public void registerUser(String username, String email) {
       // register user
   }
}
```

If you want to write a unit test for the `UserService` class that focuses on testing the `registerUser` method, how would you do it? Right now this class has two dependencies, and if you include both, you're no longer writing a unit test, you're writing an integration test.

This is where a mocking framework comes into play. Instead of using the production `UserRepository`, you will use a mock of that class; the mock *looks* like the class but isn't. You've probably heard the phrase "it acts like a duck, quacks like a duck, but isn't a duck." This mock will look like a `UserRepository` but isn't one and won't perform the *actual* business logic like persisting a new user to a database. Instead, it will do whatever you program the mock to do.

Focus only on testing your `UserService` class; the dependencies will be tested in their own unit tests. In this example, we use a popular mocking framework for the Java world named Mockito, but again the concepts should translate to whatever language you're using.

With Mockito in place, you can write a `UserServiceTest`:

```
@ExtendWith(MockitoExtension.class)
class UserServiceTest {

   @Mock
   private UserRepository userRepository;
   @Mock
   private EmailService emailService;
   @InjectMocks
   private UserService userService;

   @Test
   public void testUserRegistration() {
       String username = "newuser";
       String email = "newuser@example.com";
   when(userRepository.existsByUsername(username)).thenReturn(false);
       userService.registerUser(username, email);
       verify(userRepository).save(any(User.class));
       verify(emailService).sendWelcomeEmail(email);
   }

}
```

In the preceding test code, mock objects are created using the `@Mock` annotation to simulate dependencies, which allows for controlled testing without needing real implementations. These simulated dependencies are then automatically injected into the class being tested.

Writing Integration Tests

In the previous section, you saw an example with `UserService`, which depended on both `UserRepository` and `EmailService`. When writing a unit test, you wanted to isolate that class and thus mocked out the two dependencies.

In an integration test, you want to test how `UserService` interacts with its dependencies. Start by creating a new test named `UserServiceIntTest`. The `Int` as part of the name clearly identifies this test as an integration test.

`UserService` depends on both the repository and email service, so you will need instances of each of those classes to create an instance of the `UserService`. In a real-world application, you might be using a framework with dependency injection, allowing you to just ask for an instance, but here you will need to create them manually:

```
class UserServiceIntTest {
   private UserRepository userRepository;
   private EmailService emailService;
   private UserService userService;

   @BeforeEach
   void setUp() {
       userRepository = new UserRepository();
       emailService = new EmailService();
       userService = new UserService(userRepository,emailService);
   }

}
```

Now that you have an instance of `UserService` that contains real (not mocked) versions of your dependencies, you can write some integration tests:

```
@Test
void shouldNotRegisterUserWithUsernameOfUser() {
   // Test valid user registration
}

@Test
void shouldRegisterValidUser() {
   // Test invalid user registration
}
```

Writing End-to-End Tests

E2E testing focuses on the backend API. These tests require a real environment to run, in this case, Spring Boot with an embedded Tomcat server. The goal is to ensure that a real server is running, sending requests, and validating responses. Here's a concise example of an E2E test (the full example can be found on GitHub):

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureMockMvc
class UserRegistrationE2ETest {

   @Autowired private MockMvc mockMvc;
   @Autowired private UserRepository userRepository;
   @Autowired private EmailService emailService;
   @Autowired private ObjectMapper objectMapper;

   @BeforeEach
   void setUp() {
       userRepository.deleteAll();
   }

   @Test
   void testUserRegistration() throws Exception {
       String username =
               "testuser";
       String email = "testuser@example.com";
       User request = new User(username, email);

       // Perform the request
       mockMvc.perform(post("/api/users/register")
              .contentType(MediaType.APPLICATION_JSON)
              .content(objectMapper.writeValueAsString(request)))
              .andExpect(status().isOk());

       // Verify that the user was saved in the database
       Optional<User> savedUser = userRepository.findByUsername(username);
       assertTrue(savedUser.isPresent());
       assertEquals(email, savedUser.get().email());

       // // Additional verifications (welcome email sent) omitted for brevity
   }
}
```

This test does quite a bit:

1. Configures a random port for the server
2. Uses `MockMvc` for server-side Spring MVC test support
3. Cleans the database before each test

4. Simulates a user registration request
5. Verifies the response and database state

This approach allows you to test the entire flow from API request to database interaction, ensuring that your system functions correctly end to end.

Wrapping Up

Throughout this chapter, you've explored the world of automated testing and its crucial role in software development. You've seen how tests serve as valuable documentation, improve code quality, and act as a safety net for refactoring. From unit tests to end-to-end tests, each type plays a vital part in ensuring that your software works as intended.

Remember, just like maintaining a healthy lifestyle, writing good software is about establishing positive habits and making consistent, healthy choices. Coding with a robust test suite is like having a trainer and a well-defined exercise plan. Tests give you the confidence to move forward, try new things, and make changes without the constant fear of slipping up. This confidence empowers you to navigate the complexities of software development, enabling continuous improvement and adaptation to new requirements without fear of breaking existing functionality.

As you move forward in your career, strive to make testing a habit. It might seem like extra work at first, but the long-term benefits to your code quality, project maintainability, and professional growth are immeasurable. Remember, quality isn't a one-time act—it's a consistent practice. By embracing automated testing, you're not just catching bugs; you're becoming a more effective, confident, and skilled developer.

Putting It into Practice

Testing isn't a skill that is perfected overnight; it is developed through consistent practice. The following practices offer various entry points into the world of automated testing, whether you're working on a new feature, fixing a bug, or improving a codebase. Don't feel overwhelmed by trying to implement all of these at once. Instead, choose one or two practices that resonate with your current situation and start there. As you become comfortable with these habits, gradually incorporate others. Remember, even writing a single test today is better than writing none. Here are some practices to help you build your testing skills:

Learn from open source libraries

Explore your favorite open source library:

- Read through its tests.
- What do you learn about the library's functionality that you didn't know before?
- How do their testing practices differ from yours?
- Consider adopting some of their testing strategies in your own projects.

Use tests as documentation

Take these steps when assigned to work on an unfamiliar part of the codebase:

- Look for existing tests first.
- If tests are missing or inadequate, write new ones as you learn about the code.

Improve existing codebases

Take these steps if you find your project has an inadequate number of tests:

- Don’t try to fix everything at once. Instead, make incremental improvements.
- Challenge each developer (including yourself) to write one new test a day.

Start with tests for new features

The next time you’re assigned a new feature, begin by writing tests. This practice will help you clarify requirements and design better code from the start.

Address bugs with tests

When tackling a bug, do the following:

- First, write a test that verifies the bug exists. This test should fail initially.
- As you’re writing the tests, this might present an opportunity to refactor the code to improve the quality, readability, or maintainability.

Advocate for testing

Do the following if you find an issue that could have been prevented with proper tests:

- Add tests that would have caught the problem.
- Use this as a learning opportunity for the team.
- Discuss with your manager how improved test coverage could prevent similar issues in the future.

Practice

The only way to get better at writing tests is to practice. Consider practicing with code katas to build your testing skills in a structured, iterative way.

Remember, building a robust test suite is an ongoing process. Each small step you take toward better testing practices contributes to the overall quality and maintainability of your codebase.

Additional Resources

- Single responsibility principle
- Mockito
- JUnit 5
- AssertJ
- Hamcrest
- *Clean Code: A Handbook of Agile Software Craftsmanship* by Robert C. Martin (Pearson, 2008)

[1] It's even rarer for the documentation to be up-to-date and accurate.

[2] Likely on every check-in as part of a continuous integration process.

[3] The maintainers of the language and framework are responsible for testing their code, not you.