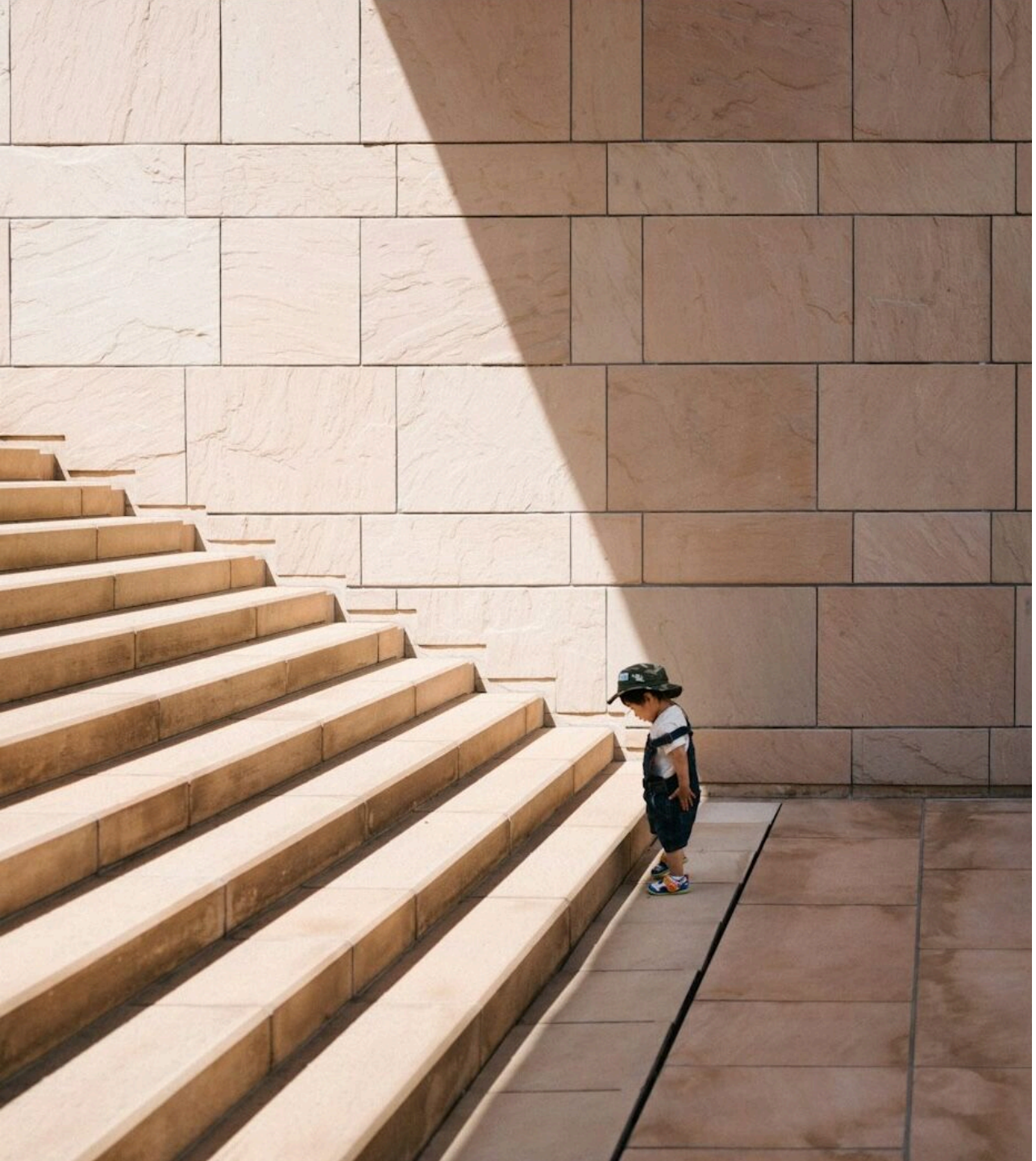

# Testing Spring Boot Applications Demystified

## Best Practices, Common Pitfalls, and Real-World Strategies

*Java User Group Zürich 21.10.2025*

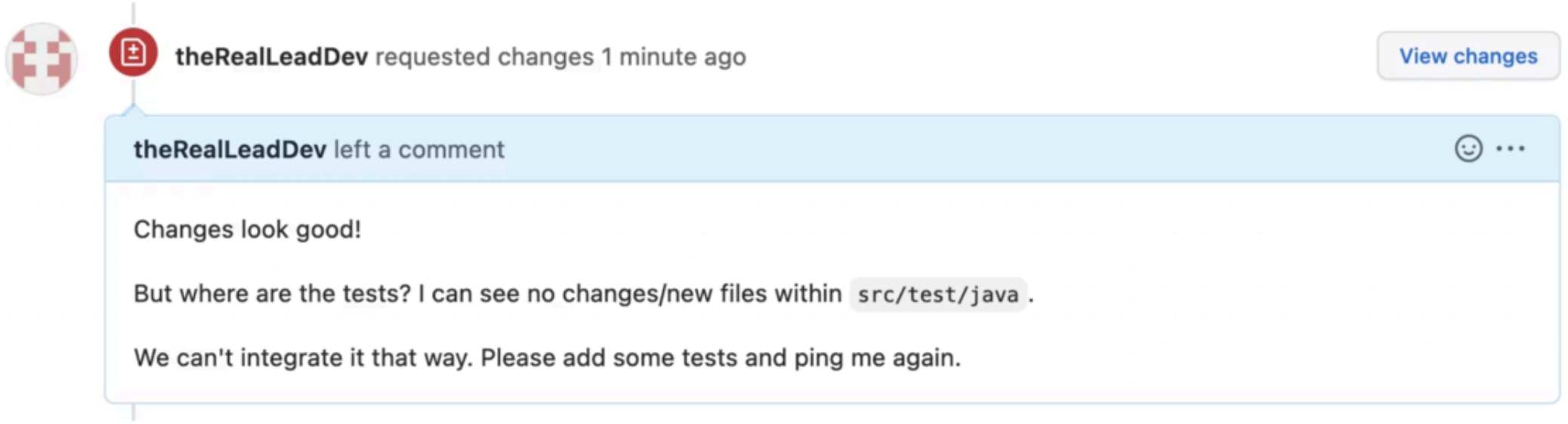
Philip Riecks - PragmaTech GmbH - @rieckpil



# Getting Started with Testing

## How It Started

# Getting Used To Testing At Work



# Goals For This Talk

- Lay the foundation for your Spring Boot testing success
- Introduction to Spring Boot's excellent test support
- Showcase a mix of best practices and early pitfalls
- Convince you that testing is not an afterthought

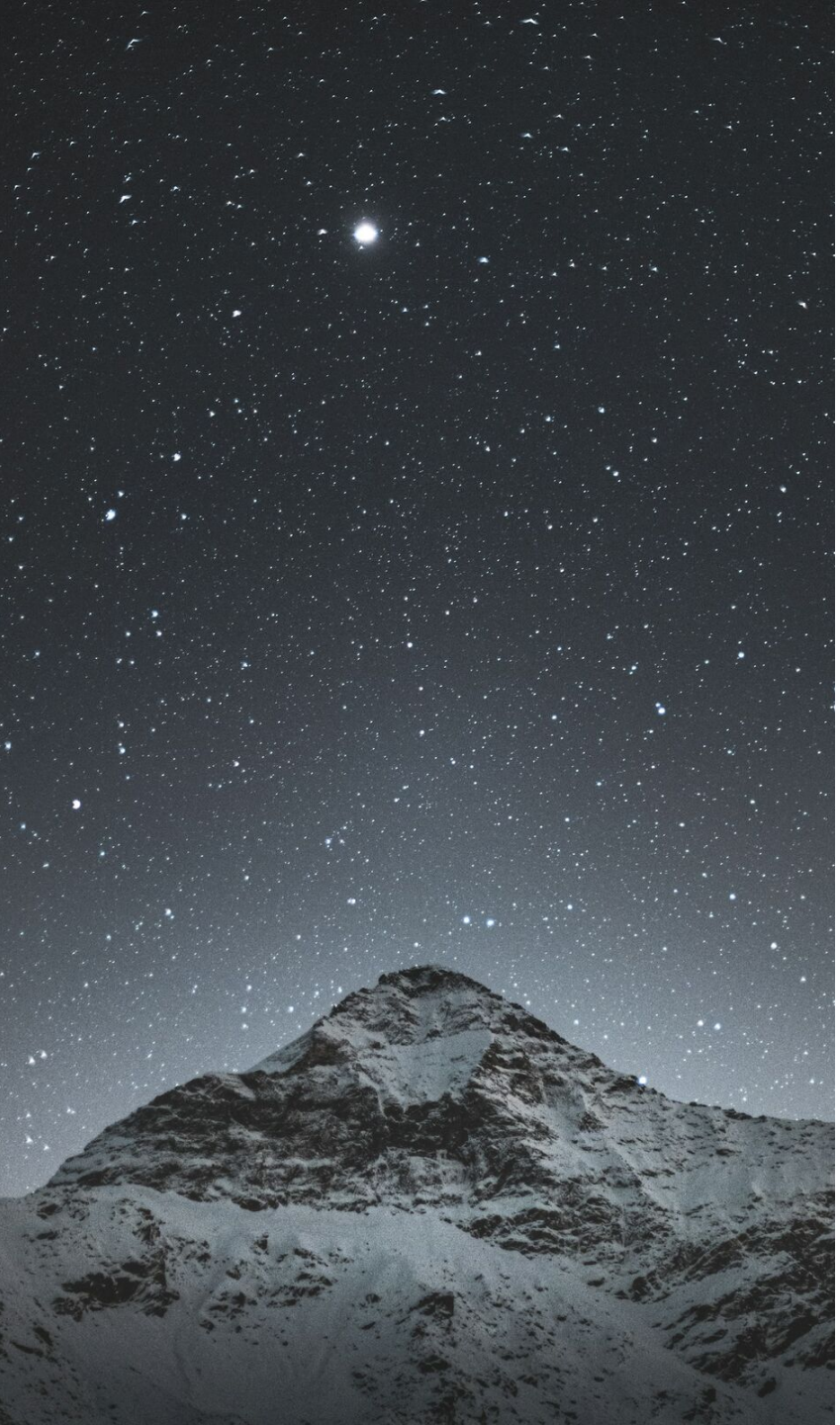
# About Philip

- Self-employed developer from Herzogenaurach, Germany (Bavaria) 🍻
- Blogging & content creation with a focus on testing Java and specifically Spring Boot applications 🍃
- Founder of PragmaTech GmbH - Enabling Developers to Frequently Deliver Software with More Confidence 🚤
- Enjoys writing tests 🧪
- @rieckpil on various platforms

# Agenda

- Introduction
- Why Test Software?
- Testing with Spring Boot
  - Spring Boot Testing 101 & Unit Testing
  - Sliced Testing
  - Integration & E2E Testing
- Spring Boot Testing Best Practices
- Common Spring Boot Testing Pitfalls
- Summary & Outlook

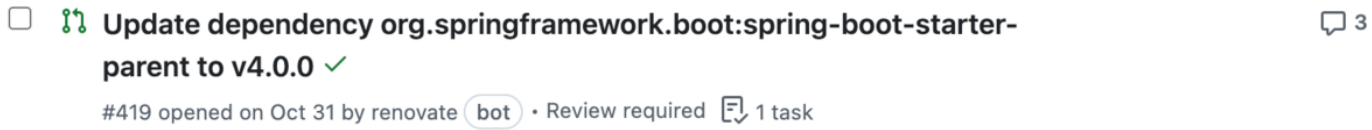
# Why Test Software?





# My Overall Northstar

Imagine seeing this pull request on a Friday afternoon:




How confident are you to merge this major Spring Boot upgrade and deploy it to production once the pipeline turns green?

Good tests don't just catch bugs - they give you the confidence to say "yes" without hesitation.

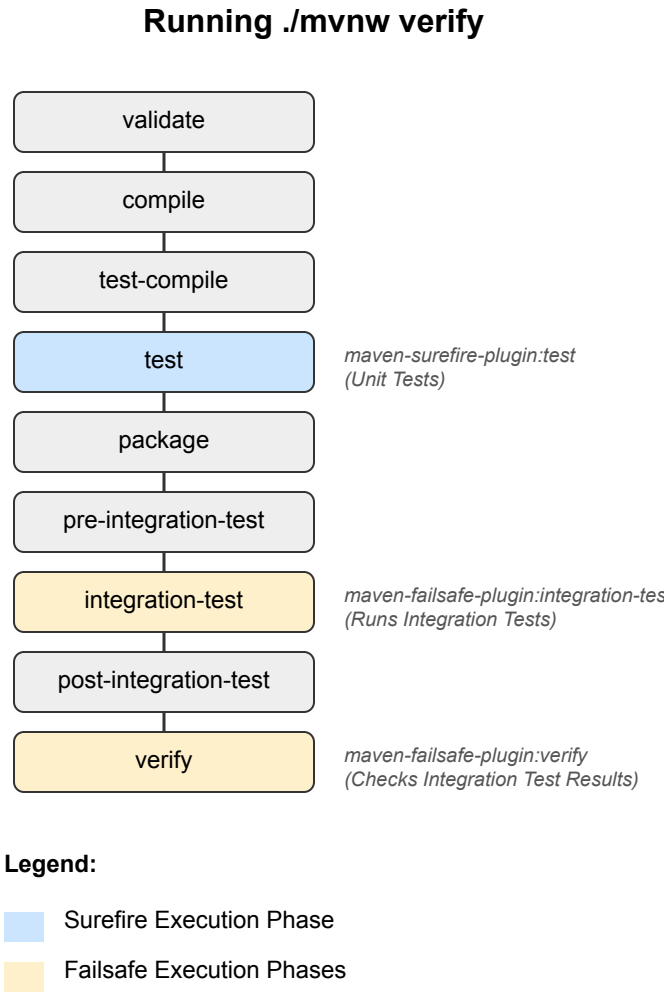
# Why Test Software? (continued)

- **Shift Left** - Catch issues earlier than the customers
- **Confidence in Code Changes** - Help new team members to onboard faster
- **Catch Bugs Early** - Reduce the ($) cost of bugs in production
- **Documentation** - Single point of truth for implemented business logic
- **Regression Prevention** - Prevent existing functionality from breaking
- **Become more Productive** - Enable faster development cycles
- **Use it as a Playground** - Explore new technologies via tests

# Spring Boot Testing 101 & Unit Testing


# Maven Build Lifecycle

- **Maven Surefire Plugin** for unit tests: default postfix `*Test` (e.g. `CustomerTest`)
- **Maven Failsafe Plugin** for integration tests: default postfix `*IT` (e.g. `CheckoutIT`)
- Reason for splitting: different **parallelization** options, better **organisation**



## Gradle Build Lifecycle

- Unit tests are run during the `test` task
- To separate integration tests, we need a custom Gradle task, as this structure is **not part** of default Gradle lifecycle
- We need to configure the `integrationTest` task manually in our `build.gradle`:

```
// Sample configuration from the Gradle docs
tasks.register('integrationTest', Test) {
  description = 'Runs integration tests.'
  group = 'verification'

  // ...
  shouldRunAfter test

  useJUnitPlatform()
}
```

# Spring Boot Starter Test

- aka. "Testing Swiss Army Knife"

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

- Batteries-included for testing by transitively including popular testing libraries
  - JUnit
  - Mockito
  - Assertion libraries: AssertJ, Hamcrest, XMLUnit, JSONAssert, Awaitility

```
./mvnw dependency:tree
[INFO] ...
[INFO] +- org.springframework.boot:spring-boot-starter-test:jar:3.5.6:test
[INFO] |  +- org.springframework.boot:spring-boot-test:jar:3.5.6:test
[INFO] |  +- org.springframework.boot:spring-boot-test-autoconfigure:jar:3.5.6:test
[INFO] |  +- com.jayway.jsonpath:json-path:jar:2.9.0:test
[INFO] |  +- jakarta.xml.bind:jakarta.xml.bind-api:jar:4.0.2:test
[INFO] |  |  \- jakarta.activation:jakarta.activation-api:jar:2.1.4:test
[INFO] |  +- net.minidev:json-smart:jar:2.5.2:test
[INFO] |  |  \- net.minidev:accessors-smart:jar:2.5.2:test
[INFO] |  |     \- org.ow2.asm:asm:jar:9.7.1:test
[INFO] |  +- org.assertj:assertj-core:jar:3.27.4:test
[INFO] |  |  \- net.bytebuddy:byte-buddy:jar:1.17.7:test
[INFO] |  +- org.awaitility:awaitility:jar:4.3.0:test
[INFO] |  +- org.hamcrest:hamcrest:jar:3.0:test
[INFO] |  +- org.junit.jupiter:junit-jupiter:jar:5.12.2:test
[INFO] |  |  +- org.junit.jupiter:junit-jupiter-api:jar:5.12.2:test
[INFO] |  |  |  +- org.junit.platform:junit-platform-commons:jar:1.12.2:test
[INFO] |  |  |  \- org.apiguardian:apiguardian-api:jar:1.1.2:test
[INFO] |  |  +- org.junit.jupiter:junit-jupiter-params:jar:5.12.2:test
[INFO] |  |  \- org.junit.jupiter:junit-jupiter-engine:jar:5.12.2:test
[INFO] |  |     \- org.junit.platform:junit-platform-engine:jar:1.12.2:test
[INFO] |  +- org.mockito:mockito-core:jar:5.16.0:test
[INFO] |  |  +- net.bytebuddy:byte-buddy-agent:jar:1.17.7:test
[INFO] |  |  \- org.objenesis:objenesis:jar:3.3:test
[INFO] |  +- org.mockito:mockito-junit-jupiter:jar:5.16.0:test
[INFO] |  +- org.skyscreamer:jsonassert:jar:1.5.3:test
[INFO] |  |  \- com.vaadin.external.google:android-json:jar:0.0.20131108.vaadin1:test
[INFO] |  +- org.springframework:spring-core:jar:6.2.11:compile
[INFO] |  |  \- org.springframework:spring-jcl:jar:6.2.11:compile
[INFO] |  +- org.springframework:spring-test:jar:6.2.11:test
[INFO] |  \- org.xmlunit:xmlunit-core:jar:2.10.4:test
```

# What's Inside the Testing Swiss Army Knife?

- **JUnit** (currently 5, later 6): Java's de-facto standard testing framework and foundation.
- **Mockito**: Creating mock objects to simulate dependencies and verify interactions.
- **AssertJ**: Provides fluent, chainable, and readable assertions.
- **Hamcrest**: Offers flexible matchers for creating custom assertions.
- **JSONAssert**: Compares JSON strings with flexible matching options.
- **JsonPath**: Extracts and queries data from JSON similar to XPath.
- **XMLUnit**: Compares and validates XML documents.
- **Awaitility**: Handles asynchronous testing with fluent conditions.

# Unit Testing Spring Boot Applications 101

- **Core Concept**: Test individual components (classes, methods) in complete isolation from their dependencies.
- **Confidence Gained**: Provides logarithmic verifications, ensuring that the smallest parts of your code work as expected under various conditions.
- **Best Practices**: Focus on a single unit of work.
- **Pitfalls**: Requires a well-thought-out class design. Poor design can lead to testing overly complex "god classes," making tests difficult to write and maintain.
- **Tools**: JUnit (or Spock, TestNG, etc.), Mockito and assertion libraries like AssertJ or Hamcrest.

# Unit Testing with Spring Boot

- Provide collaborators from outside (dependency injection) -> no `new` inside our code
- Develop small, single responsibility classes
- Test only the public API of our class
- Verify behavior not implementation details
- TDD can help design (better) classes

# Check the Imports

- Nothing Spring-related here
- Rely only on JUnit, Mockito and an assertion library

```
 1 import org.junit.jupiter.api.DisplayName;
 2 import org.junit.jupiter.api.Nested;
 3 import org.junit.jupiter.api.Test;
 4 import org.junit.jupiter.api.extension.ExtendWith;
 5 import org.junit.jupiter.params.ParameterizedTest;
 6 import org.junit.jupiter.params.provider.CsvSource;
 7 import org.mockito.Mock;
 8 import org.mockito.junit.jupiter.MockitoExtension;
 9
10 import static org.assertj.core.api.Assertions.assertThat;
```

# Unify Test Structure

- Use a consistent test method naming: givenWhenThen, shouldWhen, etc.
- Structure test for the Arrange/Act/Assert test setup

```
1  @Test
2  void should_When_() {
3
4    // Arrange
5    // ... setting up objects, data, collaborators, etc.
6
7    // Act
8    // ... performing the action to be tested on the class under test
9
10   // Assert
11   // ... verifying the expected outcome
12 }
```

```java
@ExtendWith(MockitoExtension.class)
class CustomerServiceTest {

  @Mock
  private CustomerRepository customerRepository;

  @InjectMocks
  private CustomerService customerService;

  @Test
  void shouldCreateNewCustomerWhenNameDoesNotExist() {

    when(customerRepository.findByCustomerName("duke"))
      .thenReturn(Optional.empty());

    when(customerRepository.save(any(CustomerEntity.class)))
      .thenAnswer(invocation -> {
        CustomerEntity storedCustomer = invocation.getArgument(0);
        storedCustomer.setId("42");
        return storedCustomer;
      });

    String customerId = customerService.createNewCustomer("duke");

    assertThat(customerId).isEqualTo("42");
  }
}
```

# Unit Testing Has Limits

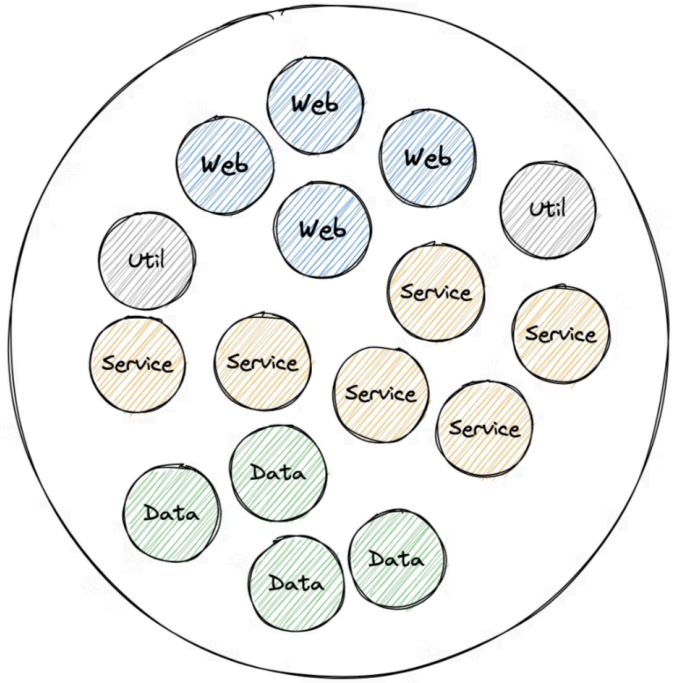
Writing a unit test for our web layer ( `UserController` ) might not cover all aspects:

- **Request Mapping**: Does `/api/users/{id}` actually resolve to our desired method?
- **Validation**: Will incomplete request bodys result in a 400 bad request or return an accidental 200?
- **Serialization**: Are we JSON objects serialized and deserialized correctly?
- **Headers**: Are we setting `Content-Type` or custom headers correctly?
- **Security**: Are we Spring Security configuration and other authorization checks enforced?

# Sliced Testing

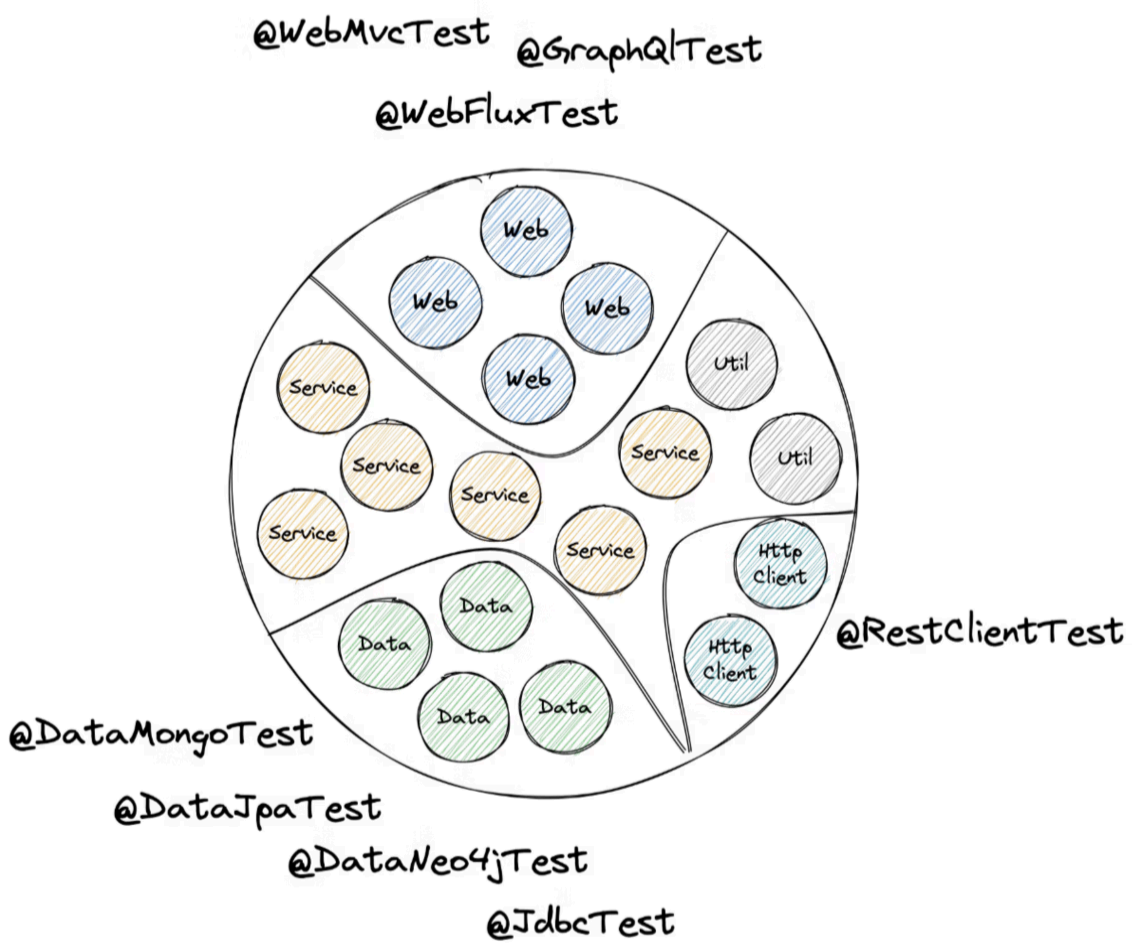
# A Typical Spring `ApplicationContext`

Our application context consists of many different components (Spring beans):



# We Can Slice It!

Spring Boot allows to load only specific parts (slices) of the application context:

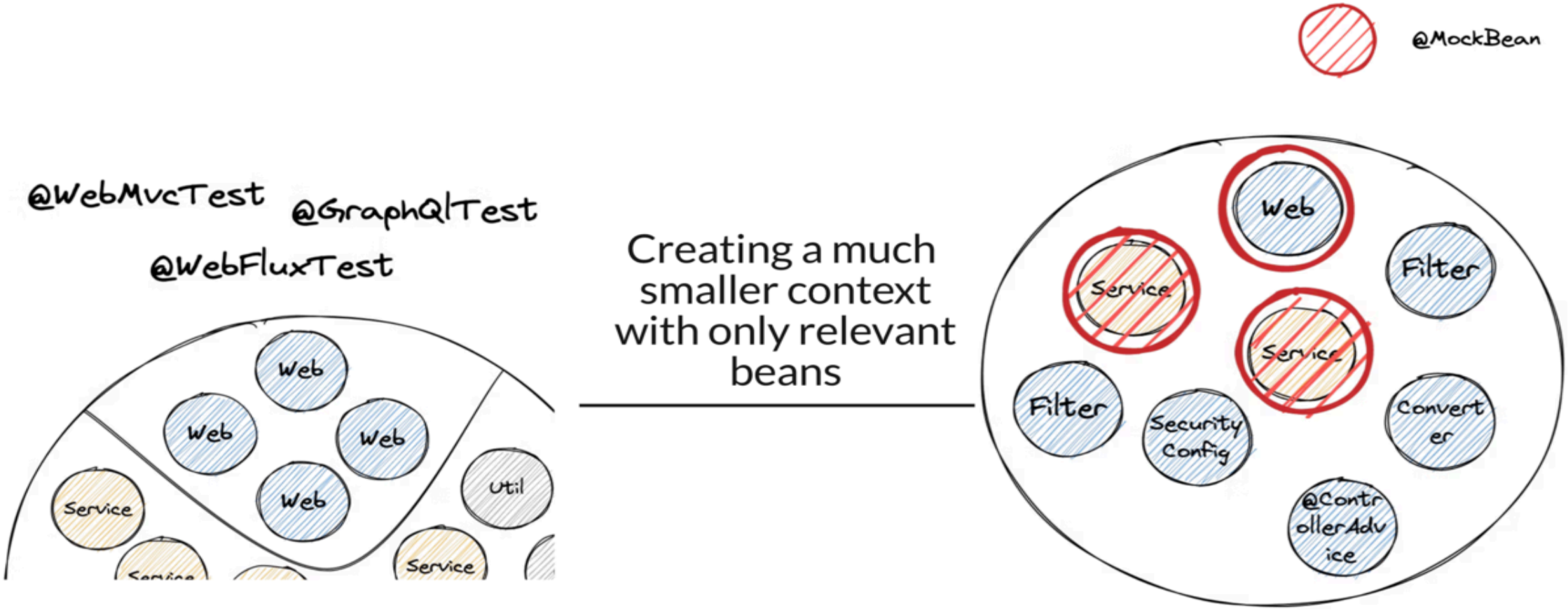


# Sliced Testing Spring Boot Applications 101

- **Core Concept**: Test a specific "slice" or layer of your application by loading a minimal, relevant part of the Spring `ApplicationContext`.
- **Confidence Gained**: Helps validate parts of your application where pure unit testing is insufficient, like the web, messaging, or data layer.
- **Prominent Examples:** Web layer (`@WebMvcTest`) and database layer (`@DataJpaTest`)
- **Pitfalls**: Requires careful configuration to ensure only the necessary slice of the context is loaded.
- **Tools**: JUnit, Mockito, Spring Test, Spring Boot, Testcontainers

# Slicing in Action

We need to provide beans that are not part of the slice:



# Slicing Example: `@WebMvcTest`

- Testing the web layer in isolation and only load the beans we need
- `MockMvc` : Mocked servlet environment with HTTP semantics
- See `WebMvcTypeExcludeFilter` for included Spring beans

```
1  @WebMvcTest(CustomerController.class)
2  class CustomerControllerTest {
3
4    @Autowired
5    private MockMvc mockMvc;
6
7    @MockitoBean
8    private CustomerService customerService;
9
10 }
```

# Common Test Slices

- `@WebMvcTest` / `@WebFluxTest` - Controller layer
- `@DataJpaTest` / `@JdbcTest` - Persistence layer
- `@JsonTest` - JSON serialization/deserialization
- `@RestClientTest` - RestTemplate testing
- etc.

@DataCouchbaseTest

@DataLdapTest

@RestClientTest

@JsonTest

@JooqTest

@WebFluxTest

@DataRedisTest

@WebMvcTest

@DataMongoTest

@DataCassandraTest

@GraphQlTest

@DataNeo4jTest

@WriteYourOwn*

@DataJpaTest

@SqsTest

@JdbcTest

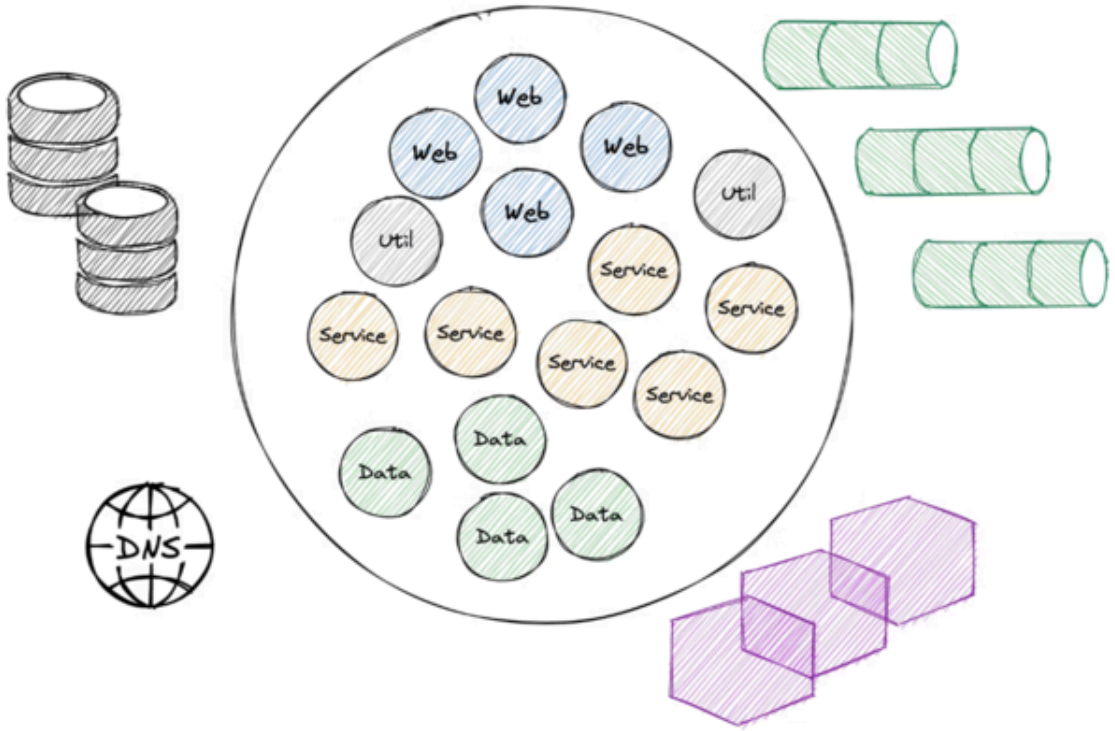
@DataElasticsearchTest

# Integration Testing

```java
@SpringBootTest
class ApplicationTest {

  @Test
  void contextLoads() {
  }

}
```



# Integration Testing Spring Boot Applications 101

- **Core Concept**: Start the entire Spring application context, often on a random local port, and test the application through its external interfaces (e.g., REST API).
- **Confidence Gained**: Validates the integration of all internal components working together as a complete application.
- **Best Practices**: Use `@SpringBootTest` to run the app on a local port.
- **Pitfalls**: Slower to run than unit or sliced tests. Managing the lifecycle of dependent services can be complex.
- **Tools**: JUnit, Mockito, Spring Test, Spring Boot, Testcontainers, WireMock (for mocking external HTTP services), Selenium (for browser-based UI testing)

# Starting the Entire Context

- Provide external infrastructure with Testcontainers
- Start Tomcat with: `@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)`
- Consider WireMock/MockServer for stubbing external HTTP services
- Test controller endpoints via: `MockMvc`, `WebTestClient`, `TestRestTemplate`
- Speed up builds with Spring Test `TestContext` caching

# Provide External Infrastructure with Testcontainers

Running infrastructure components (databases, message brokers, etc.) in Docker containers for our tests becomes a breeze with Testcontainers:

```
1 @Container
2 @ServiceConnection
3 static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:16-alpine")
4   .withDatabaseName("testdb")
5   .withUsername("test")
6   .withPassword("test")
7   .withInitScript("init-postgres.sql");
```

This gives us an ephemeral PostgreSQL database for our tests:

```
1 $ docker ps
2 CONTAINER ID   IMAGE                          COMMAND                  CREATED          STATUS          PORTS                                                NAMES
3 a958ee2887c6   postgres:16-alpine             "docker-entrypoint.s…"   10 seconds ago   Up 9 seconds    0.0.0.0:32776->5432/tcp, [::]:32776->5432/tcp        affectionate_cannon
4 ad0f804068dc   testcontainers/ryuk:0.12.0     "/bin/ryuk"              10 seconds ago   Up 9 seconds    0.0.0.0:32775->8080/tcp, [::]:32775->8080/tcp        testcontainers-ryuk-1f9f76a6-46d4-4e19-85c1-e8364da12804
```

# Stub External HTTP Services with WireMock

Consider WireMock to stub external HTTP services during tests.

- Run as in-memory service or Docker container to simulate connected HTTP services
- Simulate failures, slow responses, etc.
- Stateful setups possible (scenarios): first request fails, then succeeds
- Override HTTP clients to connect to the WireMock server during tests

```
wireMockServer.stubFor(
  get(urlPathEqualTo("/api/books/" + isbn))
    .willReturn(aResponse()
      .withHeader(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
      .withBodyFile("book-response-success.json"))
);
```

# Starting the Entire Spring Context - Version 1

- We access the application over HTTP like a user, the test and context run in separate threads (no `@Transactional` rollback), requires HTTP authentication

```
1  @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
2  class ApplicationServletContainerIT {
3
4    @LocalServerPort
5    private int port; // <-- we're running on a real port
6
7    @Test
8    void contextLoads(@Autowired WebTestClient webTestClient) {
9      webTestClient
10       .get()
11       .uri("/api/customers")
12       .header("Authorization", "Basic " + Base64.getEncoder().encodeToString("user:dummy".getBytes()))
13       .exchange()
14       .expectStatus()
15       .isOk();
16   }
17 }
```

# Starting the Entire Spring Context - Version 2

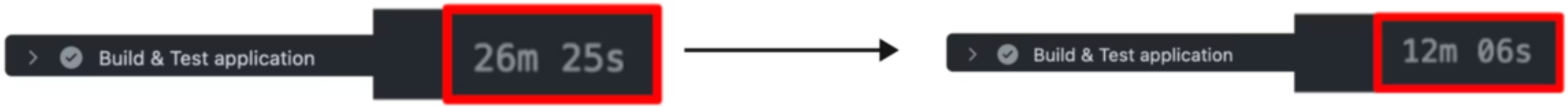
- The test and the context run in the same thread, hence we can rollback with `@Transactional` and simply override the security context with `@WithMockUser`

```
1 @SpringBootTest
2 // which is @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
3 @AutoConfigureMockMvc
4 class ApplicationMockWebIT {
5
6   // @LocalServerPort
7   // private int port; <-- this would fail the test, there is no local port occupied
8
9   @Test
10  @WithMockUser
11  void givenCustomersThenReturnListForAuthenticatedUser(@Autowired MockMvc mockMvc) throws Exception {
12    mockMvc
13      .perform(get("/api/customers")
14        .header(ACCEPT, APPLICATION_JSON))
15      .andExpect(status().is(200))
16      .andExpect(content().contentType(APPLICATION_JSON))
17      .andExpect(jsonPath("$.size()", is(1)));
18  }
19 }
```

# The Need for Speed - Reduce Build Times with Context Caching

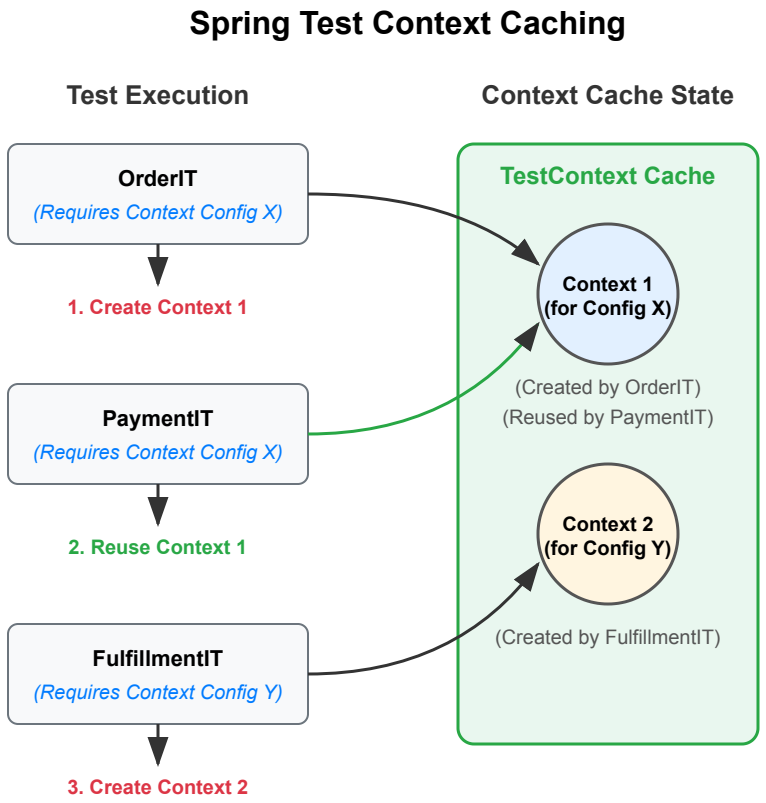
- **The problem**: Integration tests require a started & initialized Spring `ApplicationContext`, which takes time to start
- **The solution**: Spring Test `TestContext` caching, caches an already started Spring `ApplicationContext` for later reuse
- This feature is part of Spring Test (part of every Spring Boot project via `spring-boot-starter-test`)

Speed improvement example:



# Caching is King

How the caching mechanism works:



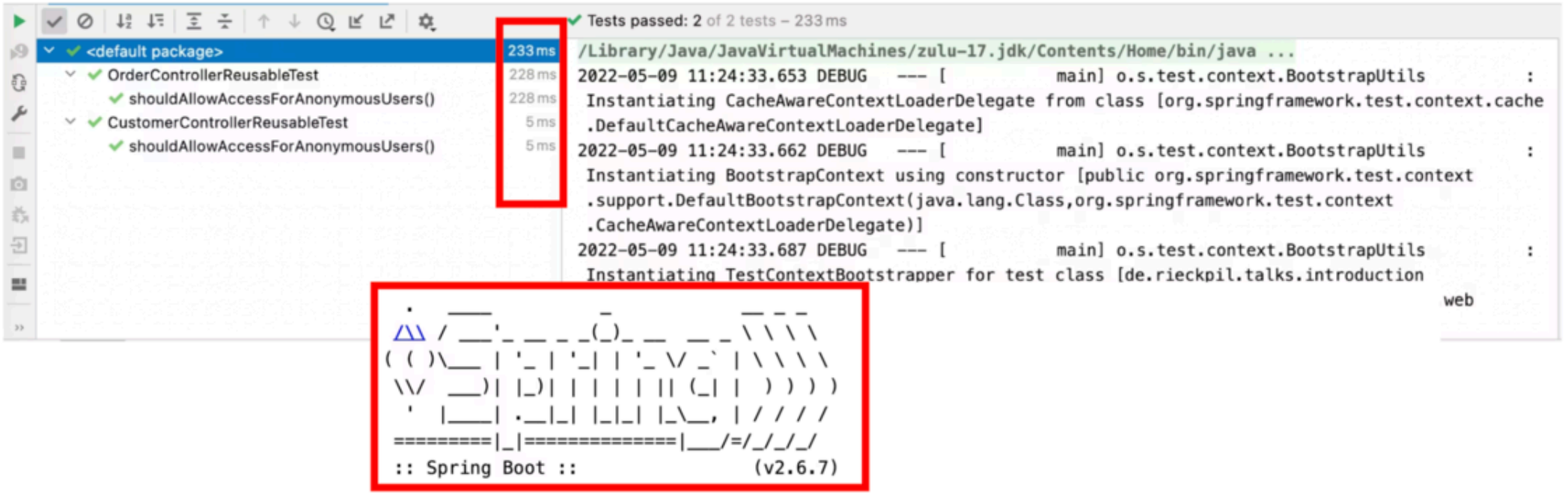
# How the Cache Key is Built

```
1 // DefaultContextCache.java
2 private final Map<MergedContextConfiguration, ApplicationContext> contextMap =
3   Collections.synchronizedMap(new LruCache(32, 0.75f));
```

This goes into the cache key ( `MergedContextConfiguration` ):

- activeProfiles ( `@ActiveProfiles` )
- contextInitializersClasses ( `@ContextConfiguration` )
- propertySourceLocations ( `@TestPropertySource` )
- propertySourceProperties ( `@TestPropertySource` )
- contextCustomizer ( `@MockitoBean` , `@MockBean` , `@DynamicPropertySource` , ...)

# Identify Context Restarts - Visually



# Identify Context Restarts - with Logs

```
<logger name="org.springframework.test.context" level="TRACE" />
```

```
2022-05-05 14:27:38.136 DEBUG 42500 --- [           main] org.springframework.test.context.cache   : Spring test
 ApplicationContext cache statistics: [DefaultContextCache@68e62b3b size = 1, maxSize = 32, parentContextCount = 0,
 hitCount = 11, missCount = 1]
2022-05-05 14:27:38.136 DEBUG 42500 --- [           main] tractDirtiesContextTestExecutionListener : After test class:
 context [DefaultTestContext@325bb9a6 testClass = ApplicationIT, testInstance = [null], testMethod = [null], testException
  = [null], mergedContextConfiguration = [WebMergedContextConfiguration@1d12b024 testClass = ApplicationIT, locations =
 '{}', classes = '{class de.rieckpil.talks.Application}', contextInitializerClasses = '[]', activeProfiles = '{}',
 propertySourceLocations = '{}', propertySourceProperties = '{org.springframework.boot.test.context
 .SpringBootTestContextBootstrapper=true, server.port=0}', contextCustomizers = set[org.springframework.boot.test.context
 .filter.ExcludeFilterContextCustomizer@5215cd9a, org.springframework.boot.test.json
 .DuplicateJsonObjectContextCustomizerFactory$DuplicateJsonObjectContextCustomizer@9257031, org.springframework.boot.test
```

# Identify Context Restarts - with Tools


An open-source Spring Test utility that provides visualization and insights for Spring Test execution, with a focus on Spring context caching statistics.

**Overall goal**: Identify optimization opportunities in your Spring Test suite to speed up your builds and ship to production faster and with more confidence.

# The Final Boss

Developers tend to consult AI/StackOverflow for integration test issues and often copy advice from the internet without knowing the implications:

```
1 @SpringBootTest
2 @DirtiesContext
3 // this instructs Spring to remove the context from the cache
4 // and rebuild a new context on every request
5 public abstract class AbstractIntegrationTest {
6
7 }
```

The setup above will **disable** the context caching feature and slow down the builds significantly!

# Spot the Issues for Context Caching

```java
@DirtiesContext
@Testcontainers
@ActiveProfiles("integration-test")
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
abstract class AbstractIntegrationTest {
```

What's "bad" for context caching here?

```java
@ActiveProfiles("integration-test")
@Import(SomeTestConfiguration.class)
@ContextConfiguration(initializers = CustomInitializer.class)
@SpringBootTest(properties = {"features.login-enabled=true", "custom.message=duke42"})
class ShowcaseIT {

  @MockBean
  private OrderService orderService;

  @SpyBean
  private CustomerService customerService;

  @Test
  void shouldInitializeContext(@Autowired ApplicationContext applicationContext) {
    assertThat(applicationContext)
            .isNotNull();
  }
}
```

# Outlook to Spring Framework 7: Pausing of Test Contexts

See the release notes of Spring Framework 7.0.0 M7.

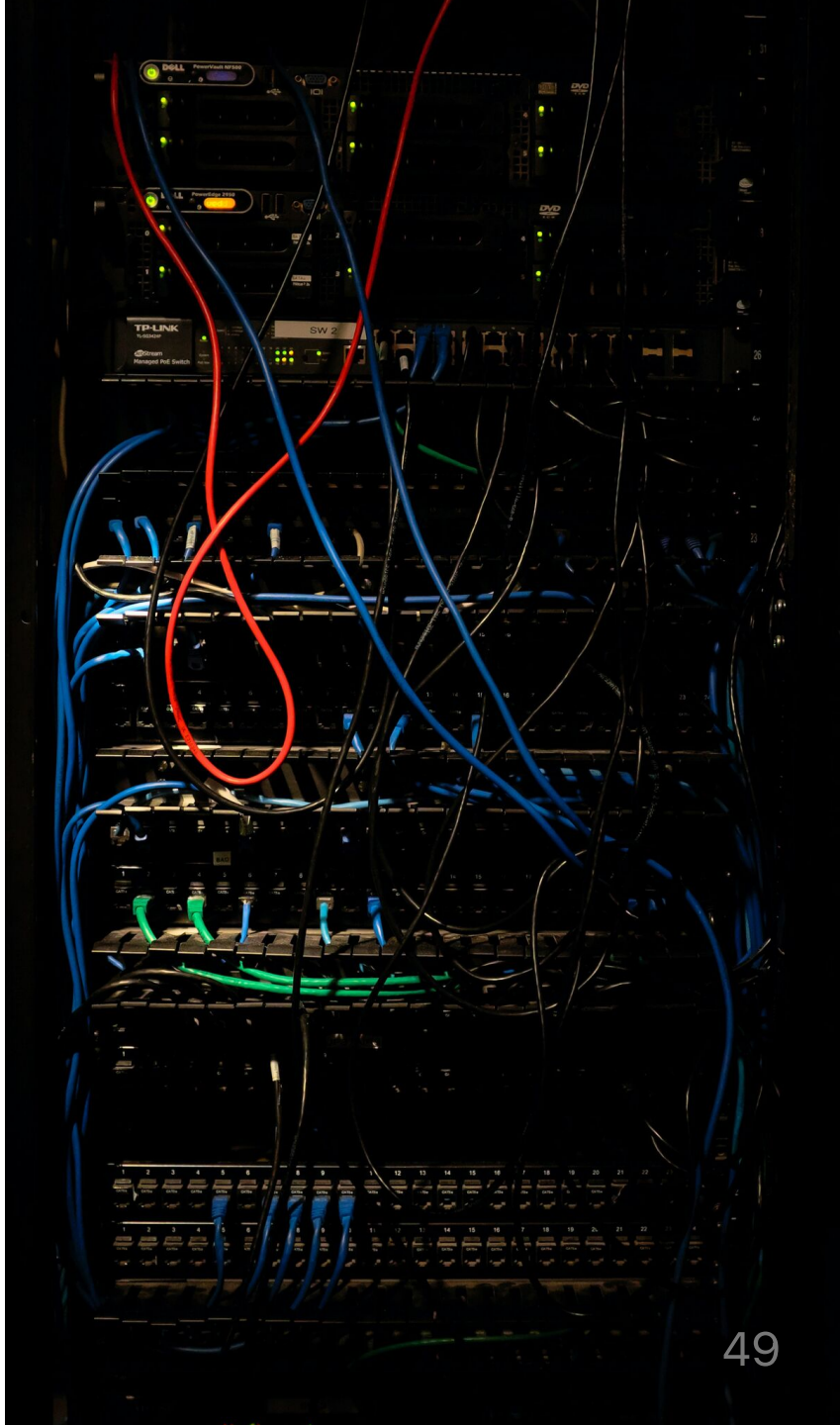
> Pausing of Test Application Contexts
>
> The Spring TestContext framework is caching application context instances within test suites for faster runs. As of Spring Framework 7.0, we now pause test application contexts when
> they're not used.
>
> This means an application context stored in the context cache will be stopped when it is no longer actively in use and automatically restarted the next time the context is retrieved from the cache.
>
> Specifically, the latter will restart all auto-startup beans in the application context, effectively restoring the lifecycle state.

# Make the Most of the Caching Feature

- Avoid `@DirtiesContext` when possible, especially central places
- Understand how the cache key is built
- Monitor and investigate the context restarts
- Align the number of unique context configurations for your test suite

# E2E Testing - the Holy Grail of Confidence

- For applications involving a UI consider tools like Selenium, Selenide, Cypress, Playwright, etc.
- Detect issues that only appear in production-like environments, also for downstream systems
- Start with a QA/DEV environment
- Consider Canary Testing and run your E2E tests regularly with a cron-like setup
- Challenges: authentication, test data management, environment stability, flakiness



# Spring Boot Testing Best Practices

## Best Practice 1: Test Parallelization

**Goal**: Reduce build time and get faster feedback

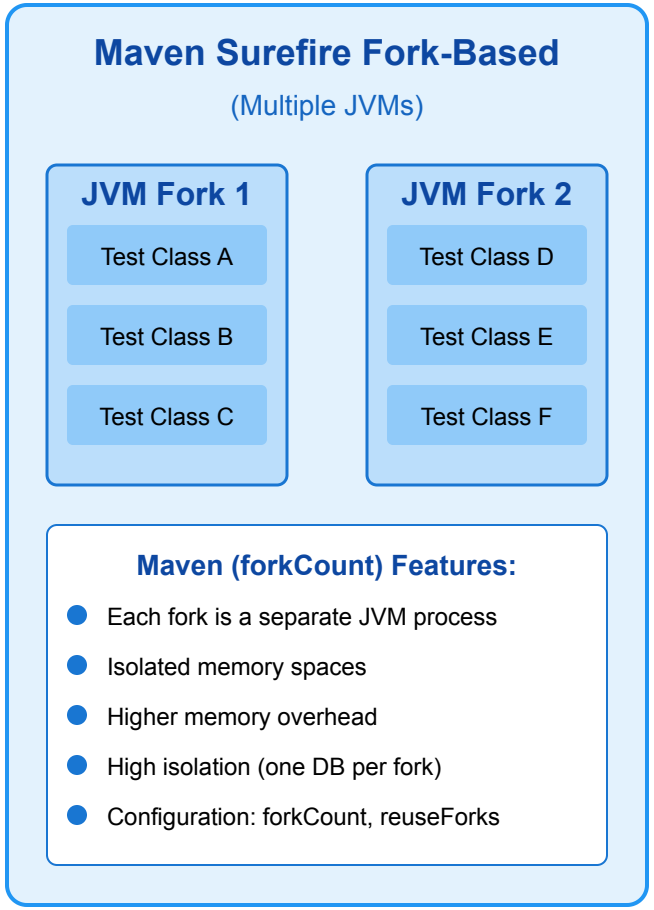
Requirements:

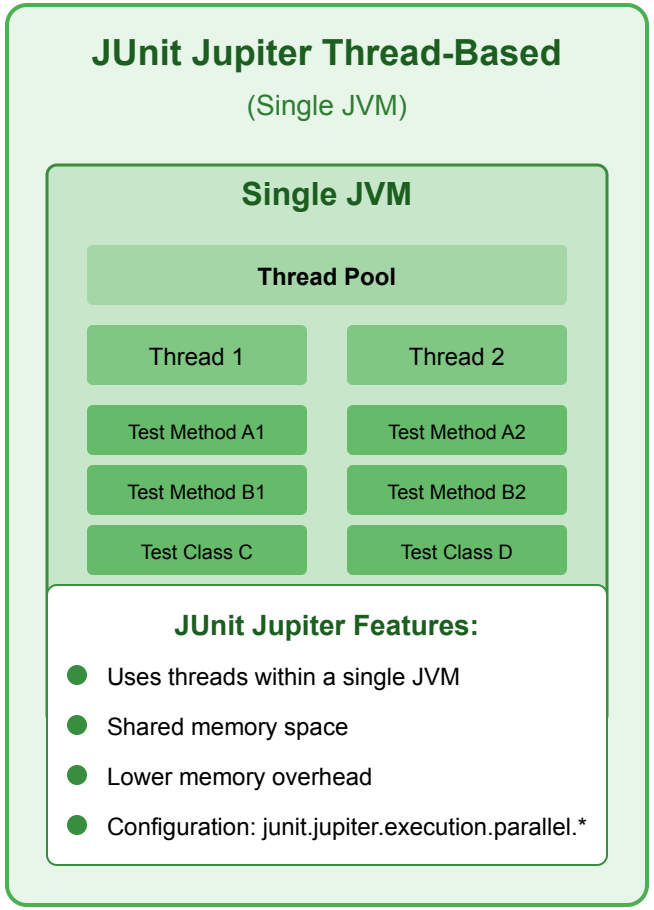
- No shared state
- No dependency between tests and their execution order
- No mutation of global state

Two ways to achieve this:

- Fork a new JVM with Surefire/Failsafe and let it run in parallel -> more resources but isolated execution
- Use JUnit Jupiter's parallelization mode and let it run in the same JVM with multiple threads

# Java Test Parallelization Options



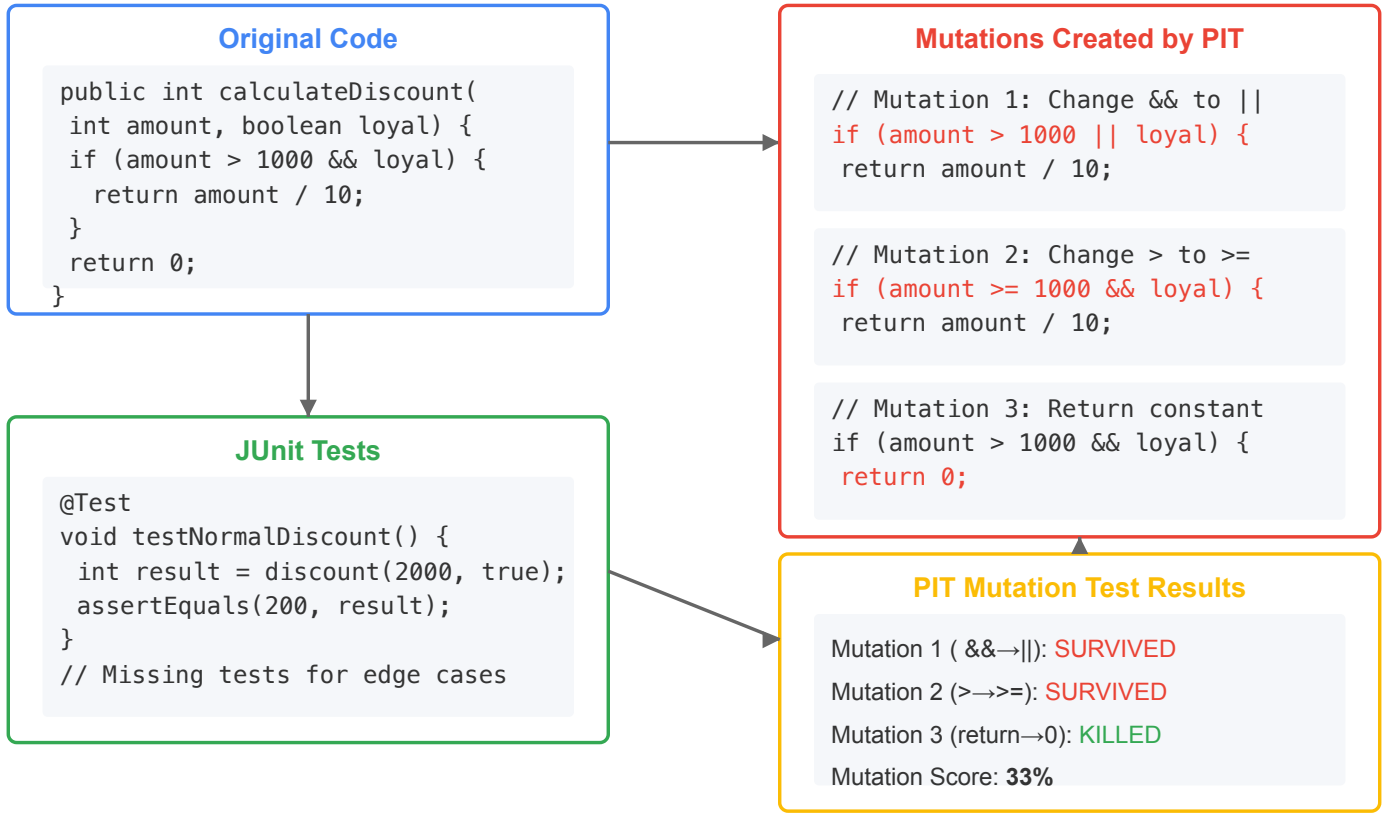
## Best Practice 2: Get Help from AI

- Diffblue Cover: AI Agent for unit testing complex (Spring Boot) Java code at scale
- My go-to CLI code agent: Claude Code
- TDD with an LLM?
- (Not AI but still useful) OpenRewrite for automatic code migrations (e.g. JUnit 4 -> JUnit 5)
- Clearly define your requirements in e.g. `claude.md` or Cursor rule files to adopt a common test structure

## Best Practice 3: Try Mutation Testing

- Having high code coverage might give you a **false sense of security**
- Mutation Testing with PIT
- Beyond Line Coverage: Traditional tools like JaCoCo show which code runs during tests, but PIT verifies if our tests actually detect when code behaves incorrectly by introducing "**mutations**" to our source code.
- Quality Guarantee: PIT automatically **modifies our code** (changing conditionals, return values, etc.) to ensure our tests fail when they should, **revealing blind spots** in seemingly comprehensive test suites.

# PIT Mutation Testing Example



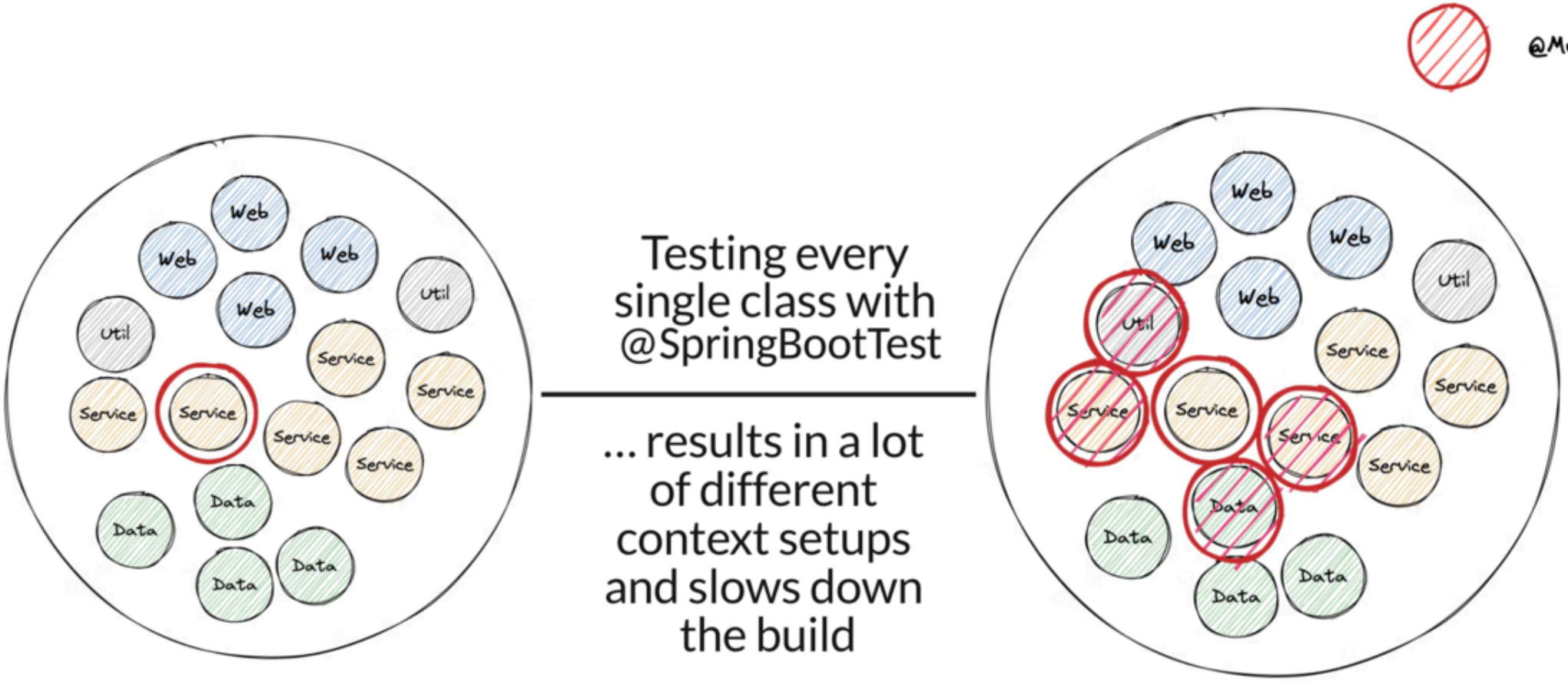

Action Required: Add tests for boundary cases (1000, loyal) and different combinations of inputs

# Common Spring Boot Testing Pitfalls to Avoid

# Testing Pitfall 1: `@SpringBootTest` Obsession

- The name could apply it's a one size fits all solution, but it isn't
- It comes with costs: starting the (entire) application context
- Useful for integration tests that verify the whole application but not for testing a single service in isolation
- Start with unit tests, see if sliced tests are applicable and only then use `@SpringBootTest`

# @SpringBootTest Obsession Visualized



# Testing Pitfall 2: @MockitoBean vs. @MockBean vs. @Mock

- `@MockBean` is a Spring Boot specific annotation that replaces a bean in the application context with a Mockito mock
- `@MockBean` is deprecated in favor of the new `@MockitoBean` annotation
- `@Mock` is a Mockito annotation, only for unit tests
- Golden Mockito Rules:
  - Do not mock types you don't own
  - Don't mock value objects
  - Don't mock everything
  - Show some love with your tests

# Testing Pitfall 3: JUnit 4 vs. JUnit 5

- You can mix both versions in the same project but not in the same test class
- Browsing through the internet (aka. StackOverflow/blogs/LLMs) for solutions, you might find test setups that are still for JUnit 4
- Easily import the wrong `@Test` and you end up wasting one hour because the Spring context does not work as expected

| JUnit 4 | JUnit 5 |
| --- | --- |
| @Test from org.junit | @Test from org.junit.jupiter.api |
| @RunWith | @ExtendWith/@RegisterExtension |
| @ClassRule/@Rule | - |
| @Before | @BeforeEach |
| @Ignore | @Disabled |
| @Category | @Tag |

# Summary & Outlook

- Spring Boot applications come with batteries-included for testing
- Spring and Spring Boot provides many excellent testing features
- Java provides a mature & rich testing ecosystem
- Consider the context caching feature for fast builds
- Get help from AI
- Still many new testing-related features are part of new releases: pausing a `TestContext`, `@ServiceConnection`, Testcontainers support, Docker Compose support, more AssertJ integrations, etc.

# What's Next?

- Online Course: Testing Spring Boot Applications Masterclass (on-demand, 12 hours, 130+ modules)
- eBook: 30 Testing Tools and Libraries Every Java Developer Must Know
- eBook: Stratospheric - From Zero to Production with AWS
- Spring Boot testing workshops (in-house/remote/hybrid)
- Consulting offerings, e.g. the Test Maturity Assessment for projects/teams


## Don't Leave Empty-Handed

- Get the complementary **Testing Spring Boot Applications Demystified eBook** for free
- 120+ Pages with hands-on advice to ship code with confidence
- Scan the **QR code on the next slide** to get the free eBook by joining our newsletter

# Joyful Testing!

Get the Spring Boot Testing eBook here:

Reach out any time via:

- LinkedIn (Philip Riecks)
- X (@rieckpil)
- Mail (philip@pragmatech.digital)