Building the Persistence Layer

Esteban Herrera

Author

@eh3rrera | eherrera.net

Choosing a Database for Unit Testing

Fake databases

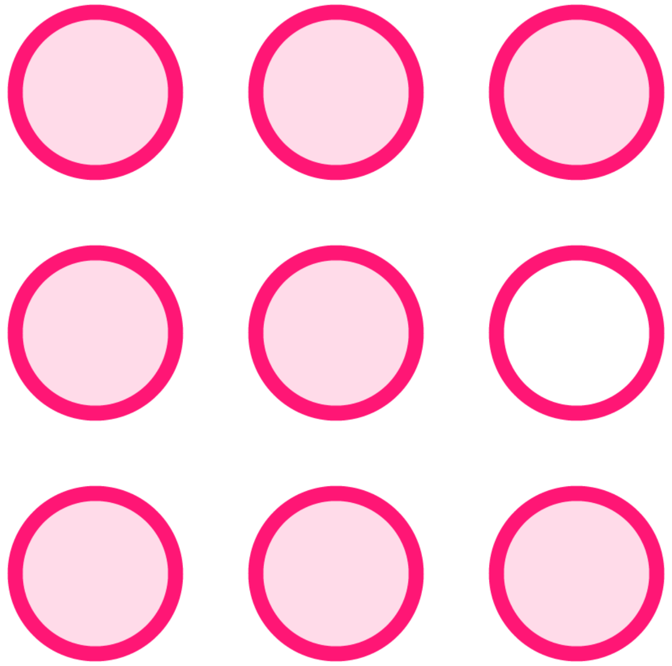
- Simple data structures (e.g., maps, lists)

In-memory databases

Real databases

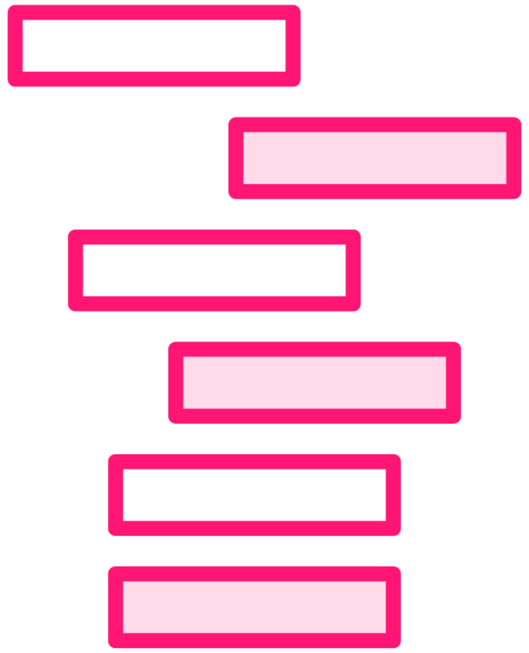
- Testcontainers

Databases in Unit Tests and Integration Tests



Unit tests

Fake databases
(emphasis on database interface interaction)



Integration tests

Embedded or real databases
(for verifying the interaction with a real database)

Fake Databases

Pros

- Very fast
- You can mock very specific behaviors from your real database

Cons

- You are essentially writing a mini-database engine
- You will not capture database-specific behaviors, constraints, or optimizations

Best for

- Unit tests where you need to mock certain behaviors or error conditions
- Simple applications with basic database interactions

In-memory Databases

Pros

- Fast setup and teardown
- Broad compatibility

Cons

- They might not always replicate the behavior of actual databases
- Not ideal for testing native SQL queries or database-specific features

Best for

- Simple operations where database-specific features are not used

Real databases or Testcontainers

Pros

- Mirror production behavior
- You can test database-specific features, queries, and transactions

Cons

- Slower test start-up time
- You need the database or Docker installed

Best for

- Integration tests where you need to mimic the production database as closely as possible
- Tests for complex SQL queries or stored procedures

Inserting Test Data

SQL Scripts
(@SQL annotation)

DatabaseRider
(DBUnit)

Manual inserts
(TestEntityManager or service/repository layer)

SQL Scripts

MyDatabaseTest.java

```java
// Spring annotations
@Sql(scripts = "/testData.sql")
public class MyDatabaseTest {
    @Test
    //@Sql(scripts = "/testData.sql")
    void testData1() {
        // ...
    }

    @Test
    void testData2() {
        // ...
    }
}
```

src/test/resources/testData.sql

```sql
INSERT INTO my_table (id, name)
VALUES (1, 'Test Data 1');

INSERT INTO my_table (id, name)
VALUES (2, 'Test Data 2');

INSERT INTO my_table (id, name)
VALUES (3, 'Test Data 3');
```

DatabaseRider (DBUnit)

MyDatabaseTest.java

```java
// Spring annotations
@DBRider
public class MyDatabaseTest {
    @Test
    @Dataset("/testData.yml")
    void testData() {
        // ...
    }
}
```

src/test/resources/testData.yml

```yaml
my_table:
  - id: 1
    name: "Test Data 1"
  - id: 2
    name: "Test Data 2"
  - id: 3
    name: "Test Data 3"
```

Manual Inserts

MyDatabaseTest1.java

```java
// Spring annotations
public class MyDatabaseTest {
    @Autowired
    private TestEntityManager entityManager;

    @Test
    void testData() {
        MyEntity entity = new MyEntity();
        // Set entity fields ...
        entityManager.persist(entity);

        // Rest of the test ...
    }
}
```

MyDatabaseTest2.java

```java
// Spring annotations
public class MyDatabaseTest {
    @Autowired
    private MyRepository myRepository;

    @BeforeEach
    void setup() {
        MyEntity entity = new MyEntity();
        // Set entity fields ...
        myRepository.save(entity);
    }
    // Test methods ...
}
```

Ensure tests don't interfere with each other.

Avoid running tests against your production database.

Spring Data JpaRepository Magic

```java
public interface UserRepository extends JpaRepository<User, Long> {
   // No additional methods needed for basic CRUD operations!
   // Methods like save(), findById(), findAll(), and delete() are inherited
   // And they're already tested by the Spring team!
}
```

When to Test Repositories?

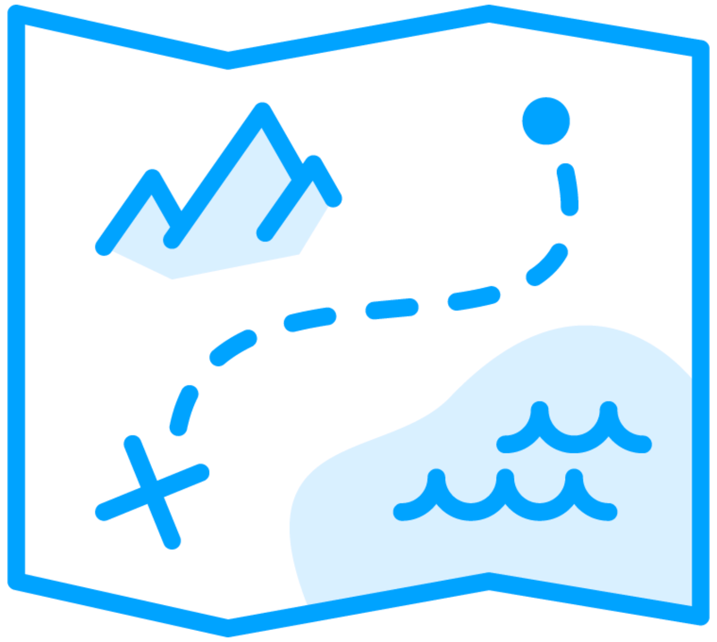
- ✓ Custom Queries
- ✓ Custom Implementations
- ✓ Transaction Behavior
- ✓ Data Integrity

```java
@DataJpaTest
public class MyRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private MyRepository myRepository;

    @Test
    void testYourRepositoryMethod() {
        // Your test logic here ...
    }
}
```

@DataJpaTest Annotation

- Limited application context
- In-memory database configuration
- Transactional tests
- SQL logging

Testing the Persistence Layer for the Demo Application



Use @DataJpaTest

In-memory H2 database

@SQL annotations

Custom method that filters tickets by criteria

Setting up the Test Class

Filtering Tickets Based on Multiple Criteria

Integration tests

Why Integration Tests Matter?

- Detect configuration and wiring issues
- Test external system integrations
- Capture regressions
- Test real environment differences

Integration Tests with TestRestTemplate

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@Sql({"/testData.sql"})
public class MyIntegrationTests {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    void test1() {
        ResponseEntity<TicketDto> response =
                      restTemplate.getForEntity("/tickets/1", TicketDto.class);

        assertEquals(HttpStatus.OK, response.getStatusCode());
    }

}
```

Integration Tests with WebTestClient

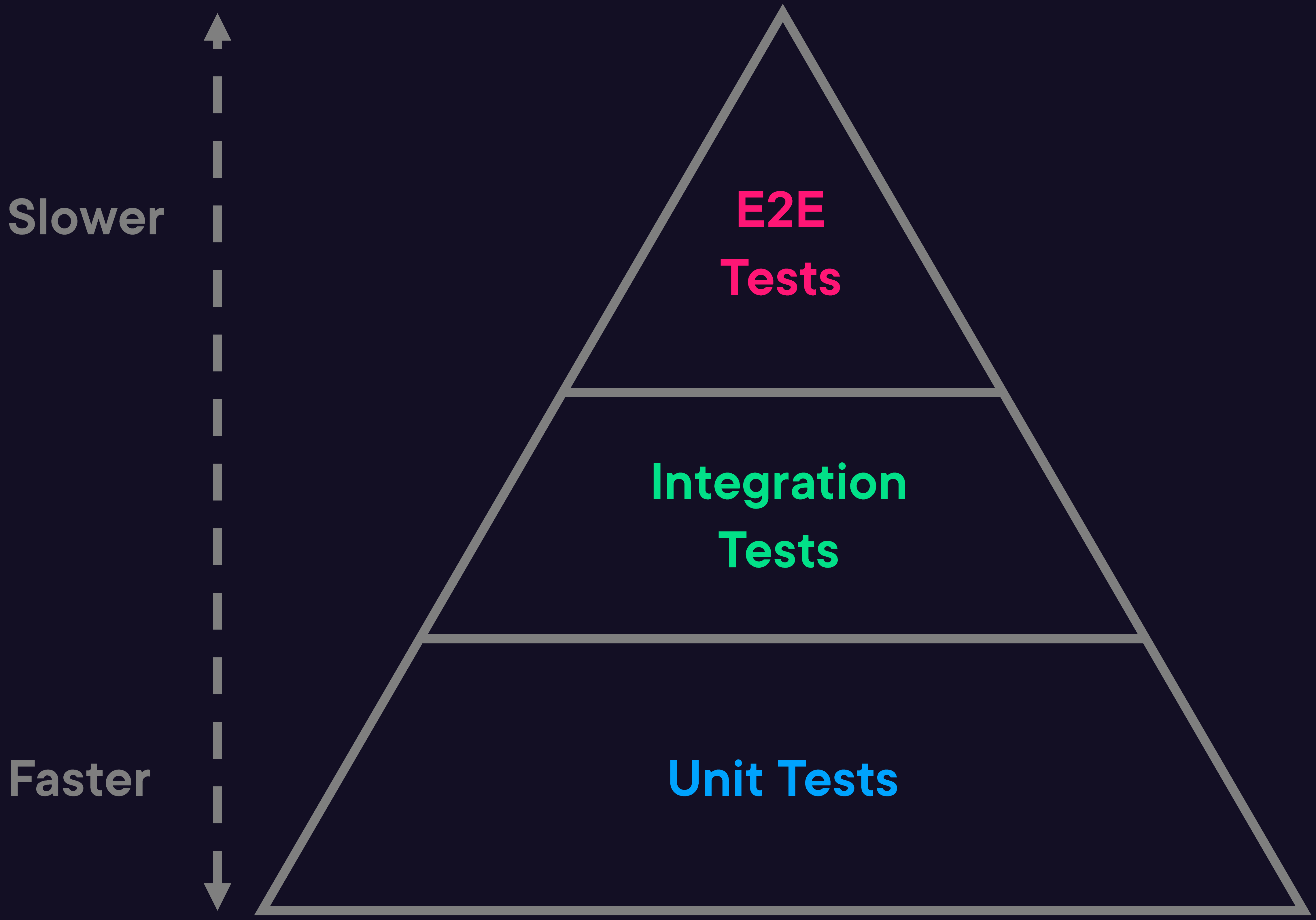
```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@Sql({"/testData.sql"})
public class MyIntegrationTests {

    @Autowired
    private WebTestClient webTestClient;

    @Test
    void test1() {
        webTestClient.get()
                  .uri("/tickets/1")
                  .exchange()
                  .expectStatus().isOk();
    }
}
```

Using WebTestClient as a Test Dependency

pom.xml

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

Testing Pyramid



Integration tests

Configuring JUnit Tags in Gradle

build.gradle

```
tasks.named('test') {
    useJUnitPlatform {
        includeTags 'MY_TAG'
    }
}
```

Summary

Unit tests vs integration tests

Red-Green-Refactor cycle

For complex applications:

- Dependency Inversion Principle
- Hexagonal Architecture

Start testing:

- Core business logic
- Presentation layer

Summary

Web layer

- Initial design
- Business rules validations in the service layer using exceptions
- Data Transfer Objects (DTOs)

Arrange/Act/Assert pattern

BDD naming style (Given/When/Then)

Summary

Service layer

- ZOMBIES acronym
- Transformation Priority Premise (TPP)
- Two testing methods:
  - Integration tests, which bring up the Spring context
  - Pure unit tests, which instantiate service classes directly and mock their dependencies
- Service classes can centralize business logic, coordinate workflows, or act as data containers
- JUnit annotations and nested test classes help manage test setup/teardown

Summary

Persistence layer

- Fake databases (maps/lists), in-memory databases, and real databases (optionally using Testcontainers)
- Insert test data using:
    - SQL scripts
    - DatabaseRider (DBUnit)
    - Manual insertion
- Write tests for for custom queries and complex operations

Integration tests complement unit tests

Thanks.