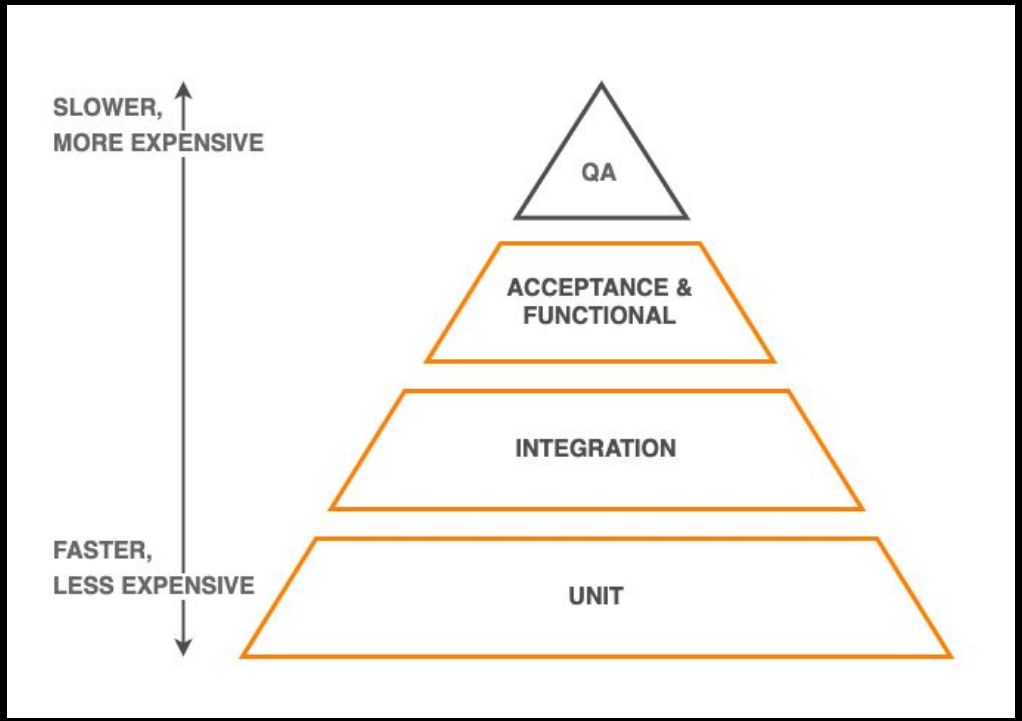
Integration Testing in Java

Ensuring Your Code Works Together

Pyramid of software testing



What is Integration Testing?

- Testing how different parts of your code interact with each other
- Ensuring that all components work together as expected
- Tests larger pieces of functionality beyond unit testing

What difference between Unit and Integration Testing?

- Result depends only on java code
- Easy to write and verify
- A single class
- All dependencies are mocked if needed
- Test verifies only implementation of code
- A unit test uses only JUnit/TestNG and a mocking framework
- A failed unit test is always a regression (if the business has not changed)

- Result depends only on external system
- Setup might be complicated
- One or more components are tested
- No mocking is used (or only unrelated components are mocked)
- Test verifies implementation of individual components and their interconnection behaviour when they are used together
- An integration test can use ***test containers*** and real DBs as well as special java integration testing frameworks (e.g. DbUnit)
- A failed integration test can also mean that the code is still correct but the environment has changed

Why is Integration Testing Important?

- Identifies issues with component interactions early on in development
- Prevents bugs and glitches from appearing later in the software lifecycle (planning /defining requirements /designing /implementing /testing /deployment).

Steps Involved in Integration Testing

- Identify components to be tested and their dependencies.
- Determine integration points between components.
- Develop test cases that will verify the interactions between the components.
- Execute test cases and analyze results.
- Identify and resolve issues that arise during testing.

What is a Java Integration Testing (mostly)

- a test uses the database
- a test uses the API

- a test uses an external system (e.g. a queue or a mail server)
- a test reads/writes files or performs other I/O

@SpringBootTest

- useful when you need to bootstrap the entire application context (check if application start)
- By default create Application context only once, but here some nuance))
- @SpringBootTest might lead to very long-running test suites

Overriding properties

- @ActiveProfiles
- @SpringBootTest(properties = {})
- @TestPropertiesSource

Examples with @SpringBootTest and test properties

@ContextConfiguration

- defines class-level metadata that is used to determine how to load and configure an ApplicationContext for integration tests
- support path-based or class-based (but mostly support only one type simultaneously)

@TestConfiguration

- If you want to customize the primary configuration, you can use a nested @TestConfiguration class. Unlike a nested @Configuration class, which would be used instead of your application's primary configuration, a nested @TestConfiguration class is used in addition to your application's primary configuration (so you can modify Spring's application context during test runtime).

@MockBean

- We can use the @MockBean to add mock objects to the Spring application context. The mock will replace any existing bean of the same type in the application context.

Test Auto-configuration Annotations

- in order not to raise the entire context as in the case of @SpringBootTest, spring provides us with tools to configure a narrower context for our tests using auto-configuration annotations

- you can read the full list of these annotations using the link in the title, in this lecture we will cover @DataJdbcTest and @WebMvcTest

@DataJdbcTest

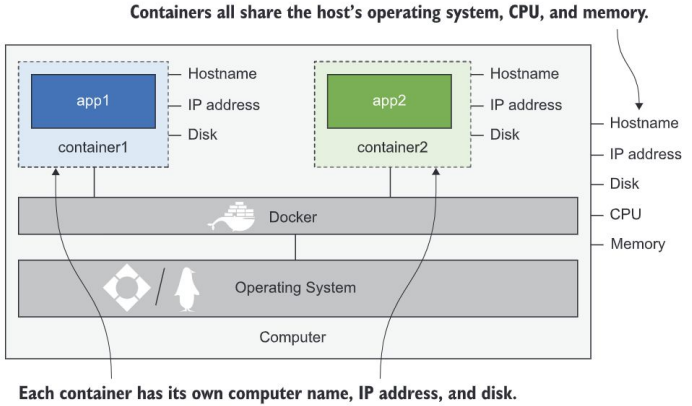
- using this annotation will disable full auto-configuration and instead apply only configuration relevant to Data JDBC tests
- by default use an embedded in-memory database (replacing any explicit or usually auto-configured DataSource)
- by default, Data JDBC tests are transactional and roll back at the end of each test

Example with @DataJdbcTest

DOCKER

- Docker is an open platform for developing, shipping, and running applications
- Docker allows you to run applications in so-called containers
- a container is a set of source code, dependencies, and settings to run your app
- containers are built on top of each other
- each container has its own virtual environment managed by the docker
- docker set ip, host name, resource and managed all this stuff
- the container runs as long as the application inside runs
- ***the container shares the operating system of the server on which it is running***

DOCKER



Containers are isolated environments, but they're super efficient. One computer can run lots of containers, which all share the same OS, CPU and memory.

Test containers

Testcontainers for Java is a Java library that supports JUnit tests, providing lightweight, throwaway instances of common databases, Selenium web browsers, or anything else that can run in a Docker container even run your own container.

Useful annotations

- @Containers - is used in conjunction with the Testcontainers annotation to indicate the containers that the Testcontainers extension should manage.
- @Testcontainers - is a JUnit Jupiter extension to activate automatic startup and stop of containers used in a test case.
- @Rule - brings up a new database for every test case in your class.
- @ClassRule - one database is brought up for all the tests in the class.

Example with test container

Responsibilities of a Web Controller

- Listen to HTTP Requests
- Deserialize Input
- Validate Input
- Call the Business Logic (unit test can do this to)
- Serialize the Output
- Translate Exceptions

@WebMvcTest

- an application context that contains only the beans needed for testing a web controller
- can take a controller class as an argument

MockMvc

- MockMvc is a Spring Boot test tool class that lets you test controllers without needing to start an HTTP server. You can test the web layer as if i’s receiving the requests from the HTTP server without the hustle of actually starting it.