



Java Technologies

Lecture 11

Testing & Deploying

Fall, 2025

Agenda

- Application Deployment Phases
- The Test Pyramid
- Unit Testing, Integration Testing
- Database Integration Testing, Testcontainers
- Performance Testing
- CI/CD Pipeline, GitHub Actions
- Process Isolation, Containerization, Docker
- Container Orchestration, Kubernetes

Application Deployment

1 Preparation

- Define deployment strategy (manual, automated, etc.)
- Prepare infrastructure (servers, containers, cloud environment)
- Ensure CI/CD pipeline is set up

2 Building, Testing, Staging

- Create deployable artifacts (JAR, WAR, Docker image, etc.)
- Run automated tests
- Validate deployment package in a staging environment.

3 Deployment

- Big Bang / Recreate
- Blue/Green Deployment
- Canary Release
- Rolling Deployment

4 Monitoring, Maintenance

- Monitor the application health, response times, logs.
- Collect feedback from users.

The systematic process of evaluating a software application to ensure it functions as intended, is free of defects, and meets all specified requirements.

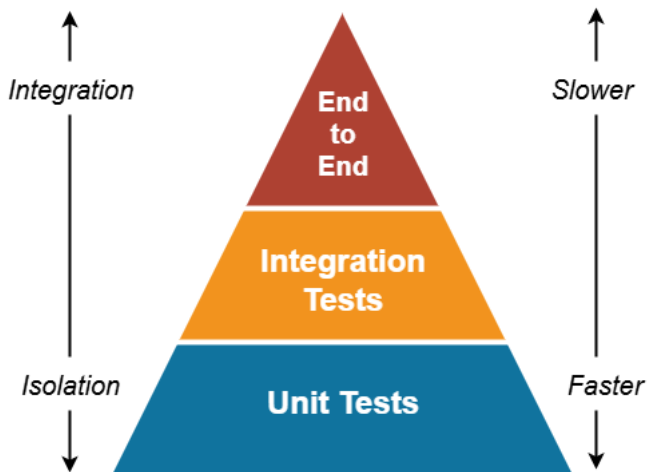
- **Functional testing**

- Unit testing
- Integration testing
- System testing
- User acceptance testing (UAT) ...

- **Non-Functional testing**

- Performance testing
- Security testing
- Usability testing
- Compatibility testing ...

The Test Pyramid



Unit Testing

- UT verifies that **each unit of code works as expected** before it is integrated with other parts of the application.
- **"Units"** are small, isolated components: methods, classes.

```
@Service
public class EquationService {

    // Solves ax + b = 0
    public double solve(double a, double b) {
        if (a == 0) {
            throw new IllegalArgumentException(
                "Coefficient 'a' cannot be zero.");
        }
        return -b / a;
    }
}
```

- A **testing framework** is used to write and run unit tests: JUnit (simple, lightweight), TestNG (advanced features)

Example: Unit Testing with JUnit 5

```
class EquationServiceTest {  
  
    // No need to use @BeforeEach  
    private EquationService equationService = new EquationService();  
  
    @Test  
    @DisplayName("Solve equation with valid coefficients")  
    void testValid() {  
        double result = equationService.solve(2, 4);  
        assertEquals(-2.0, result, 0.0001);  
    }  
  
    @Test  
    @DisplayName("Solve equation when coefficient a = 0")  
    void testAIsZero() {  
        Exception e = assertThrows(IllegalArgumentException.class,  
            () -> equationService.solve(0, 5));  
        assertTrue(e.getMessage().contains("cannot be zero"));  
    }  
} // JUnit 5 produces XML/HTML test reports when run via build tools
```

Integration Testing

- IT verifies that different components or modules of a system **work together as expected**. Examples:
 - Controller - Service communication
 - Service - Repository - Database
 - REST endpoints return expected responses
- **Big Bang**: All modules are integrated/tested at the same time.
- **Incremental**
 - **Top-Down**: Begin with the highest-level modules.
Start with the controller, use a "fake" service, move down.
Stub, Mock = temporary implementations of a lower-level module.
 - **Bottom-Up**: Begin with the lowest-level modules.
Start with the repo, use a "fake" service, then move up.
Driver = temporary placeholder of a higher-level module.
 - **Sandwich (Hybrid)**: Combines top-down and bottom-up.

Example: Controller - Service Integration

```
@RestController
@RequestMapping("/api/hello")
public class HelloController {

    private final HelloService helloService;

    public HelloController(HelloService helloService) {
        this.helloService = helloService;
    }
    @GetMapping
    public String sayHello() {
        return helloService.getMessage();
    }
}
```

```
@Service
public class HelloService {
    public String getMessage() {
        return "Hello, Spring Boot!";
    }
}
```

Example: Simple Integration Test

```
@SpringBootTest
@AutoConfigureMockMvc
class HelloControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void testSayHelloEndpoint() throws Exception {
        mockMvc.perform(get("/api/hello"))
            .andExpect(status().isOk())
            .andExpect(content().string("Hello, Spring Boot!"));
    }
}
```

- @SpringBootTest loads the full Spring context.
- @AutoConfigureMockMvc and MockMvc allow simulating HTTP requests without starting a real server.

Using a Stub: Top-Down Approach

- Assume we have the class UserService.
- We don't have a full implementation of the UserRepository.
- A **stub** simulates a fake repository **with a fixed behavior**.

```
class UserServiceStubTest {
    static class UserRepositoryStub implements UserRepository {
        @Override
        public User findById(Long id) {
            return new User(id, "Alice"); // Fake implementation
        }
        @Override
        public void save(User user) { } // Not implemented yet
    }
    @Test
    void testGetUserGreetingWithStub() {
        var stubRepo = new UserRepositoryStub();
        var userService = new UserService(stubRepo);
        String greeting = userService.getUserGreeting(1);
        assertEquals("Hello, Alice", greeting);
    }
} // We are not checking interactions, only the returned value.
```

Using Mockito to Test Interaction

- A **mock verifies interactions** between service and repository.
- The mock does not need to return a real user. By default, all methods on a mock return "empty" values.

```
class UserServiceMockTest {  
  
    @Test  
    void testSaveUserWithMock() {  
        UserRepository mockRepo = Mockito.mock(UserRepository.class);  
        UserService userService = new UserService(mockRepo); ⚠  
  
        User user = new User(1, "Bob");  
        userService.saveUser(user);  
  
        // verify that save() was called  
        // exactly once with this user  
        verify(mockRepo, times(1)).save(user);  
    }  
} // We are asserting that interactions occurred correctly.
```

Database Integration Testing

- How to run tests that check if the application and the database work together as expected?
 - Queries, ORM, Data integrity, Transactions, etc.
- The challenges:
 - **Test isolation**: Each test must run in a clean environment.
 - **Environment setup**: Setting up a database can be complex and time-consuming.
 - **Speed**: Integration tests with a DB are much slower.
 - **Data management**: Managing a consistent set of test data for a large number of tests can be difficult.
- What to do?
 - Use a dedicated test db, not a shared or the production db.
 - Use an in-memory database (like H2), for speed. ⚠
 - Use database truncation, or rollback all transactions at the end.
 - **Automate setup and teardown.**

Testcontainers Library

Open source library for providing throwaway, lightweight instances of databases, message brokers, web browsers, or just about anything that can run in a Docker container.

- **Realistic testing environment**

- Postgres, MongoDB, Kafka, Redis, etc.

- No more "works on my machine"

- **Isolated and repeatable tests**

- Each test can start with a clean container.

- **No need for local installs**

- Testcontainers will pull & run them automatically.

- **Great for CI/CD**

- Eliminates the need for dedicated test databases.

Requirements: Docker installed and running. 😬

Example: TestContainers

```
@Testcontainers
@SpringBootTest
class UserRepositoryTest {

    @Container
    static PostgreSQLContainer<?> postgres =
        new PostgreSQLContainer<>("postgres:15")
            .withDatabaseName("testdb")
            .withUsername("test")
            .withPassword("test"); // randomly assigned port


    @Autowired
    private UserRepository userRepository;

    @Test
    void shouldSaveAndFetchUser() {
        User saved = userRepository.save(new User("Alice"));
        Optional<User> fetched = userRepository.findById(saved.getId());

        assertTrue(fetched.isPresent());
    }
} // How does Spring Boot knows to use this postgres container?
```

Configuring the Test DataSource

- Usually, the DataSource is configured in application.properties

```
# Main datasource configuration (uses in-memory H2 for demo)
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.hibernate.ddl-auto=create-drop 
```

- The UserRepositoryTest can dynamically override the properties in order to use Testcontainers PostgreSQL.

```
@DynamicPropertySource
static void overrideProperties
    (DynamicPropertyRegistry registry) {
    registry.add("spring.datasource.url",
        postgres::getJdbcUrl);
    registry.add("spring.datasource.username",
        postgres::getUsername);
    registry.add("spring.datasource.password",
        postgres::getPassword);
} // What does this syntax mean postgres::getJdbcUrl ?
```

Performance Testing

- Evaluates how a system behaves in terms of **responsiveness, stability, scalability, and speed** under a particular workload.
- **Key metrics:** Response time, throughput, concurrent users, resource utilization.
- **Types of performance testing:**
 - Load testing
 - Stress testing
 - Endurance testing (or Soak testing)
 - Spike testing
 - Scalability testing
 - Volume testing
- **Load testing tools**
 - **JMeter:** Mature, GUI-heavy
 - **Gatling:** Code driven, requires Scala language
 - **K6:** Great for APIs and microservices, etc.

Example: Gatling Simulation

```
class BasicSimulation extends Simulation {  
  
  val httpProtocol = http  
    .baseUrl("http://localhost:8080") // Spring Boot URL  
    .acceptHeader("application/json")  
  
  val scn = scenario("HelloAPI Test")  
    .exec(http("Get Hello")  
      .get("/hello")  
      .check(status.is(200)))  
  
  setUp(  
    scn.inject(atOnceUsers(10)) // Simulate 10 users at once  
  ).protocols(httpProtocol)  
}
```

```
mvn gatling:test  
Response time (min/mean/max)  
Requests per second  
Success/failure rate
```

CI/CD Pipeline

CI/CD is the automated process of building, testing, and deploying code to deliver software quickly and reliably.

- **Continuous Integration (CI)**

- Commit to a shared repo (like Git)
- Linting, static analysis
- Build → executable/deployable artifact
- Automated testing

- **Continuous Delivery / Deployment (CD)**

- Deployment to staging environment
- Manual or automated end-to-end testing
- Release to production
 - Manual, one-click process
 - Automated → Continuous Deployment

- **CI/CD tools:** GitHub Actions, GitLab CI/CD, Jenkins, Azure DevOps, AWS CodePipeline, CircleCI

GitHub Actions

GitHub Actions is an event-driven CI/CD platform built into GitHub.

- **Workflow:** The primary automated process. A workflow is a configurable, multi-step process defined in a YAML file, inside your repo in the workflows directory.
- **Events:** Triggers that start a workflow (push, pull request).
- **Jobs:** A workflow is composed of one or more jobs. Each job runs in its own virtual machine or container.
- **Steps:** A job is made up of a sequence of steps. Each step can either be a shell command you write or an "action."
- **Actions:** Reusable units of code that perform a specific task: check out your code, set up a programming language, deploy to a cloud service.
- **Runners:** The servers that execute your workflows. GitHub provides hosted runners for various operating systems (Linux, Windows, macOS).

Example: Simple GitHub Workflow

Create an YAML file in your repo: `.github/workflows/lint.yml`

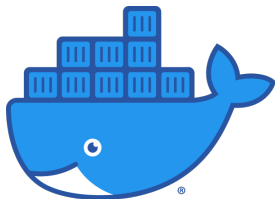
```
name: Lint Java
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code          # Step 1: Checkout the code
        uses: actions/checkout
      - name: Set up JDK 21          # Step 2: Set up JDK
        uses: actions/setup-java
        with:
          java-version: '21'
          distribution: 'temurin'
      - name: Run Checkstyle         # Step 3: Run Checkstyle (linting)
        run: mvn checkstyle:check
        continue-on-error: true
```

Process Isolation

- How to make sure that when we deploy an app on a server, it isn't influenced by other processes already running there?
- **Process isolation** is the segregation of different software processes to prevent them from accessing memory space they do not own → Each to their own virtual address space.
- But shared resources (CPU, memory, I/O, network) can still cause interference.
- A **virtual machine (VM)** is a software-based computer that runs an operating system and applications in an isolated environment on top of physical hardware through a hypervisor (VMware, Oracle Virtual Box).
 - ✓ Strong isolation & security
 - ✗ Heavy on resources, slow startup

Containerization

- **Containers** are lightweight, portable execution environments that package an application along with its dependencies, libraries, and configuration.
- **The Goal:** To make our application run in a consistent, predictable manner across different computing environments.
- Containers don't use full virtualization (like VMs), but rely on OS-level process isolation features:
 - **Namespaces:** Isolate what a process can *see*.
 - **Control groups:** Manage what a process can *use*.
 - **Union file systems:** Enable layered, efficient file storage.
- **Benefits & Trade-offs**
 - ✓ Lightweight and fast to start, efficient resource usage
 - ✗ Weaker isolation than VMs, depend on host OS



- **Docker** is an open platform for containerization.
- It popularized and standardized containers with a simple CLI, image format, and ecosystem.
 - Docker made it easy for developers.
- Runs natively on Linux, you need Docker Desktop on Windows.
- **Alternatives:** Podman, containerd, CRI-O, LXC

Dockerfile

- A Dockerfile is a text document, usually placed in the root of your project, that contains a set of instructions for building an **image of your containerized application**.
- The Docker engine reads the Dockerfile and executes its commands in a specific order, creating a series of layers that, when combined, form the final, **executable image**.

```
# Use an official OpenJDK image
FROM openjdk:21-jdk-slim

# Set working directory in container
WORKDIR /app

# Copy the jar built by Spring Boot
COPY target/myapp.jar app.jar

# Run the jar file
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Docker Terminology

- **Image:** Read-only template used to create containers.
- **Container:** Running instance of an image, isolated environment.
- **Layer:** Immutable, cached filesystem change created per Dockerfile instruction.
- **Repository:** A collection of Docker images.
- **Registry:** A service that stores images (e.g., Docker Hub).
- **Tag:** Label for image versions (myapp:1.0).
- **Entrypoint:** The command run when the container starts.
- **Build Context:** Source directory for COPY commands.
- **Volume:** Persistent storage outside the container.
- **Network:** Virtual network connecting containers.
- **Daemon:** Background service managing Docker resources.

Docker CLI

- `build`: Builds a Docker image from a Dockerfile.
- `run`: Runs a container from an image.
- `ps`: Lists running containers.
- `stop`: Stops a running container.
- `rm`: Removes a container.
- `rmi`: Removes an image.
- `pull`: Downloads an image from a registry.
- `push`: Uploads an image to a registry.
- `logs`: Shows logs from a container.
- `exec`: Runs a command inside a running container.
- `images`: Lists all downloaded images.

docker run hello-world

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that produces this output.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
`$ docker run -it ubuntu bash`

Share images, automate workflows, and more with a free Docker ID:
<https://hub.docker.com/>

For more examples and ideas, visit:
<https://docs.docker.com/get-started/>

Container Orchestration

- Consider an application using microservices.
- Each microservice can be packaged into a Docker container, creating individual images.
- How do you manage a large number of containers, each representing a different service? 😞
- How do you:
 - Find services and distribute traffic efficiently?
 - Scale containers up or down based on demand?
 - Track the health and availability of containers?
 - Collect and analyze performance data across services?
- **Container orchestration** is the process of automating the deployment, management, scaling, and networking of containers.
- **Tools:** Kubernetes (K8s), Docker Swarm, HashiCorp Nomad

Kubernetes (K8)



- **Kubernetes** is the industry standard, open-source, orchestration platform – originally designed by Google.
- Works with any container runtime (Docker, containerd)
- Complex, but powerful. It offers auto-scaling, self-healing, load balancing, service discovery, and many more related tools.
- Many cloud providers offer managed Kubernetes services, such as: Google Kubernetes Engine (GKE) Amazon Elastic Kubernetes Service (EKS) Azure Kubernetes Service (AKS)

Core Concepts of Kubernetes

- **Cluster Components**

A cluster is a set of nodes (physical or virtual machines) that work together to run your applications.

- **Node / Worker:** A machine that runs containers (pods).
- **Master / Control Plane:** Manages the cluster and makes global decisions (scheduling, scaling, etc.)

- **Workload Resources**

What should run, how it should run, and how many copies.

- **Pod:** Smallest deployable unit (one or more containers).
- **Deployment:** Manages pods (running in the desired number).

- **Networking & Storage**

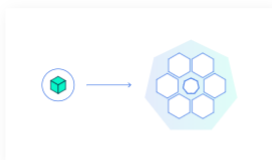
- **Service:** Exposes pods internally or externally.
- **Ingress:** Manages HTTP/HTTPS access to services.
- **PersistentVolume:** Represents storage in the cluster.

Kubernetes Flow

Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit.



1. Create a Kubernetes cluster



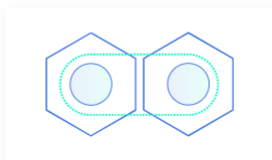
2. Deploy an app



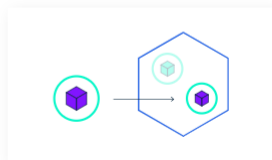
3. Explore your app



4. Expose your app publicly



5. Scale up your app



6. Update your app

<https://kubernetes.io/docs/tutorials>