Beyond Mocks: Modernizing Integration Testing with TestContainers

Mohammed Aboullaite
@laytoun
Sr Backend Engineer, Spotify
Docker Captain
Java Champion
Google Developer Expert

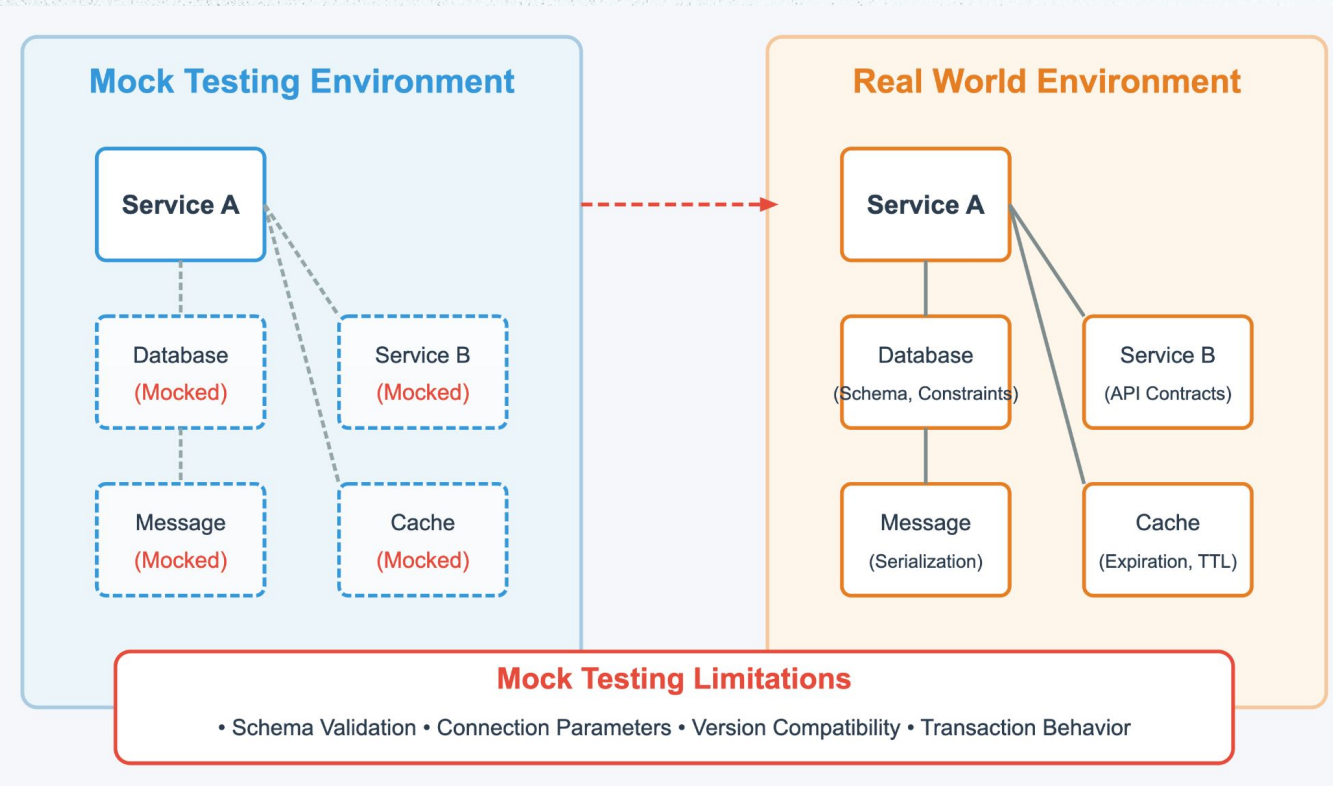
Agenda

- The Integration Testing Challenge
- Introduction to Testcontainers
- Advanced Patterns
- CI/CD Integration
- Testcontainers Cloud
- Best Practices & Pitfalls
- Q&A

The Integration Testing Challenge

- Microservices architecture complexity
- Multiple external dependencies
- Production-like environments are difficult to replicate
- Mocks only go so far

Where Mocks Fall Short



Introduction to Testcontainers

- Java library that enables using Docker containers in integration tests
- Spins up real dependency services during test execution
- Supported languages: Java, Go, .NET, Node.js, Python, and more
- Works with JUnit 4, JUnit 5, TestNG, Spock

Core Concepts

- Generic Containers
- Database Containers
- Docker Compose Support
- Singleton Containers
- Container Lifecycle Management

Key Benefits

- Tests against real services, not simulations
- Consistent environment across development and CI
- Isolated testing (no shared test databases)
- Fast startup with reusable containers
- Reduced maintenance of test environments

Demo

https://github.com/aboullaite/testcontainers-demo

Custom Containers

```
public class MinioContainer extends GenericContainer<MinioContainer> {
    private static final int DEFAULT_PORT = 9000;

    public MinioContainer() {
        super("minio/minio:latest");
        withCommand("server /data");
        withExposedPorts(DEFAULT_PORT);
        withEnv("MINIO_ACCESS_KEY", "minioadmin");
        withEnv("MINIO_SECRET_KEY", "minioadmin");
    }

    public String getEndpoint() {
        return "http://" + getHost() + ":" +
getMappedPort(DEFAULT_PORT);
    }
}
```

Container Reuse

```
// Example: Singleton Container Pattern
@Testcontainers
public class SharedPostgresqlContainer {
    public static PostgreSQLContainer<?> postgres =
        new PostgreSQLContainer<>("postgres:14")
            .withReuse(true);

    static {
        postgres.start();
    }
}

// In test classes
class Test1 {
    @BeforeAll
    static void setup() {
        System.setProperty("spring.datasource.url",

 SharedPostgresqlContainer.postgres.getJdbcUrl());
}
```

Service Discovery

```java
@Testcontainers
public class ServiceDiscoveryTest {
    @Container
    private Network network = Network.newNetwork();

    @Container
    private PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:14")
        .withNetwork(network)
        .withNetworkAliases("postgres");

    @Container
    private GenericContainer<?> service = new GenericContainer<>("myapp:latest")
        .withNetwork(network)
        .withEnv("DB_HOST", "postgres")
        .withExposedPorts(8080);

    @Test
    void testServiceIntegration() {
        // Test service that discovers Postgres via network alias
    }
}
```

CI/CD Integration - GitHub Actions

```yaml
jobs:
  test:
    runs-on: ubuntu-latest
    services:
      docker:
        image: docker:dind
        options: --privileged
    steps:
      - uses: actions/checkout@v3
      - name: Set up JDK
        uses: actions/setup-
java@v3 with:
          java-version: '17'
      - name: Run tests
        run: ./mvnw test
```

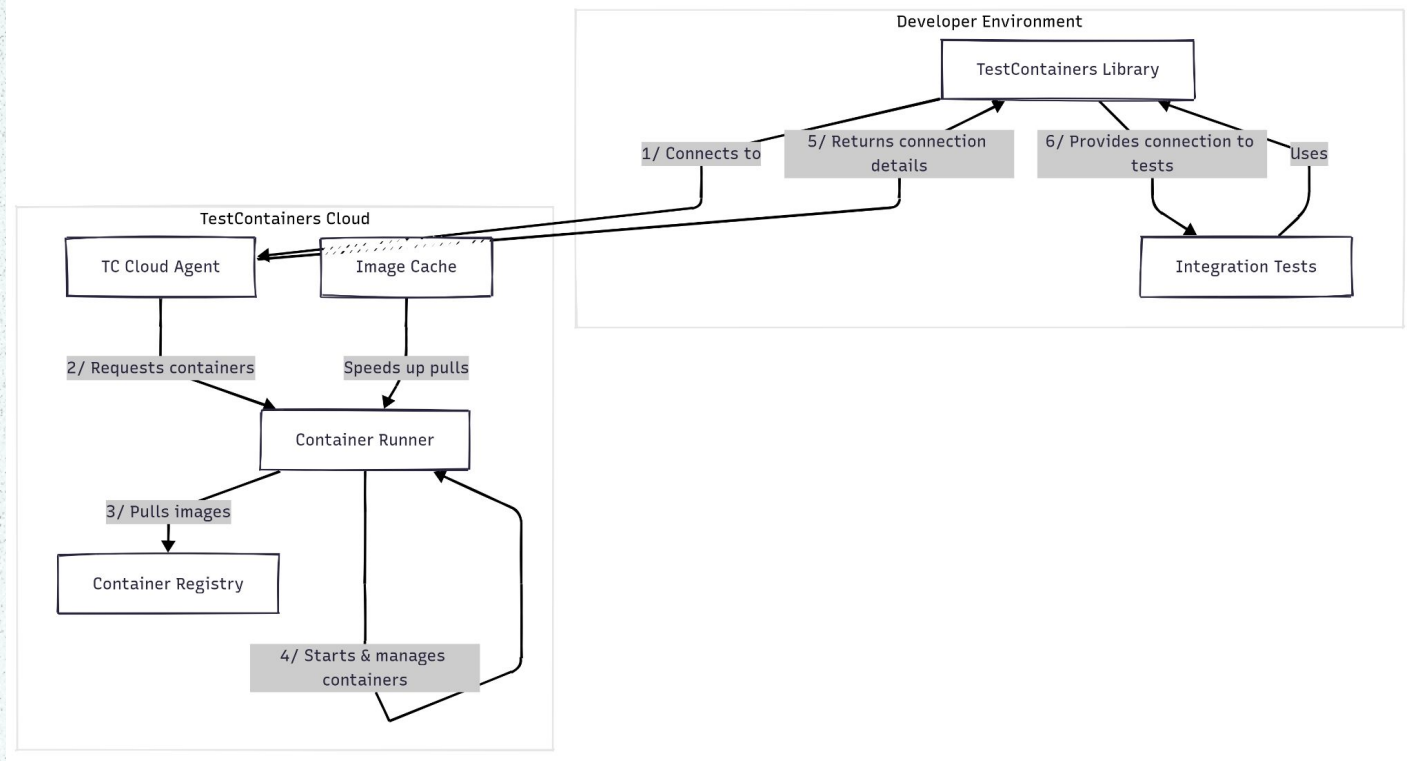
CI/CD Integration - Performance Considerations

- Container reuse strategies
- Parallelization approaches
- Resource limitations
- Caching Docker images
- Remote Docker hosts option

Introducing Testcontainers Cloud

- Docker's managed Testcontainers service
- Zero local Docker dependencies
- Runs containers in the cloud
- Seamless integration with existing tests
- Launched in 2023, now part of Docker

Testcontainers Cloud - How It Works



Testcontainers Cloud - Benefits

- No Docker installation required
- Consistent environment across all developers
- Reduced resource usage on developer machines
- Faster test execution (pre-warmed containers)
- Parallel test execution without resource constraints
- Usage insights and metrics

Best Practices

- Start containers only when needed
- Clean up all resources after tests
- Use the latest Testcontainers version
- Maintain container image versions
- Consider resource usage (CPU/memory)
- Implement wait strategies

Test Data Management

- Init scripts and migrations
- Volume mounts for test fixtures
- Programmatic data setup
- Snapshot testing approaches
- Reset strategies between tests

Common Pitfalls

- Resource leaks (containers not stopped)
- Insufficient wait strategies
- Hardcoded ports
- Ignoring container logs
- Overly complex container setups
- Using production credentials

Beyond Java - Polyglot Support

Testcontainers modules for:

- Go
- .NET
- Node.js
- Python
- Rust
- And more...

Resources

- Official documentation: `https://testcontainers.com`
- GitHub: `https://github.com/testcontainers`
- Testcontainers Cloud: `https://testcontainers.cloud`
- Sample code from this talk:
  `https://github.com/aboullaite/testcontainers-demo`

Thanks!

Do you have any questions?

@laytoun