O'REILLY®

Integration Testing with Docker and Testcontainers

About the trainer

bmuschko

bmuschko

bmuschko.com

AUTOMATED ASCENT

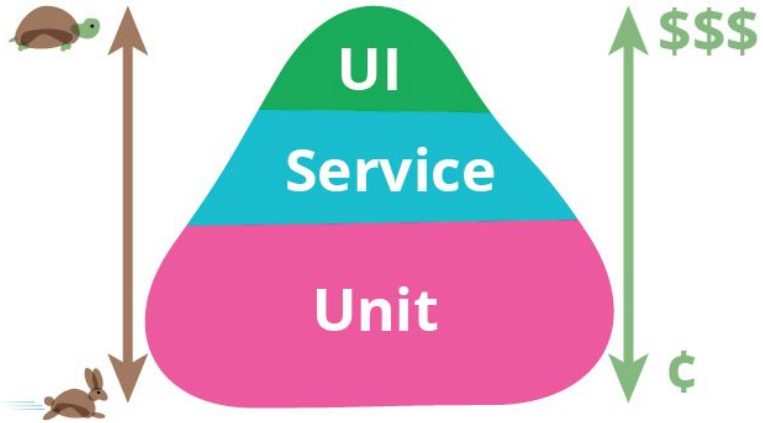
automatedascent.com

Challenges and Benefits of Integration Testing

Understanding Testcontainers and the Problems it Solves

The Testing Pyramid

Distribution, cost, and execution times per type



Blog post on Martin Fowler's blog

Integration Tests Are A Must

12 unit tests, 0 integration tests

DISCUSSION

Typical Problems with Integration Testing?

Problems with Integration Testing

Integration tests interact with other parts of the system

- Reproductible environment
- Slow startup times
- Isolated environment
- Cross-platform support

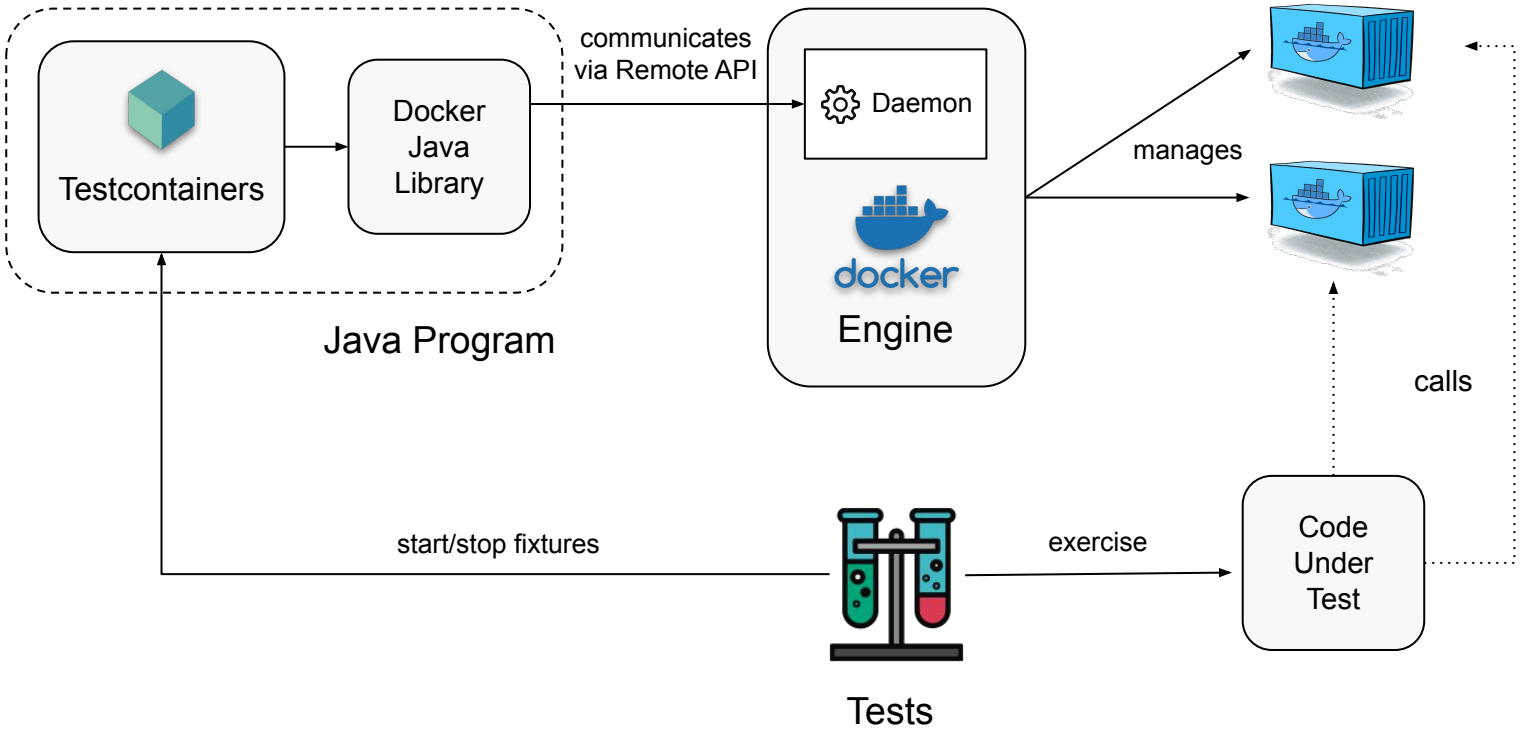
What is Testcontainers?

Library for managing containers in tests with Docker

- Creates disposable Docker containers as test fixtures
- Support for different programming languages
- Most prominent implementation is based on Java/JUnit 4 & 5

User documentation

High-Level Architecture



Interaction with Docker

Docker Java handles low-level communication

- Docker Engine communication via docker-java library
  - Doesn't require Docker executable to be installed
  - No support for buildx functionality
- Docker environment discovery
  - Detection of environment variables like `DOCKER_HOST`
  - Reads and uses credentials from `~/.docker/config.json`
- Automatic container cleanup via Moby Ryuk container

Dependencies in Maven

pom.xml

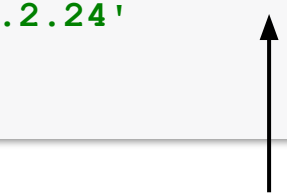
```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId>
    <version>1.16.0</version>
    <scope>test</scope>
</dependency>
```

TestContainers libraries available on Maven Central

Dependencies in Gradle

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'org.testcontainers:junit-jupiter:1.16.0'
    testImplementation 'org.testcontainers:postgresql:1.16.0'
    testRuntime 'org.postgresql:postgresql:42.2.24'
}
```



Duplicated version definition

Using the Bill of Materials

build.gradle

```
dependencies {
    testImplementation platform('org.testcontainers:testcontainers-bom:1.16.0')
    testImplementation 'org.testcontainers:mysql'
}
```

Bill of Materials (BOM)

A POM that defines compatible versions for a set of dependencies

JUnit 5 (Jupiter) Dependencies

build.gradle

```
dependencies {
    def junitJupiterVersion = '5.4.2'
    testImplementation "org.junit.jupiter:junit-jupiter-api:$junitJupiterVersion"
    testImplementation "org.junit.jupiter:junit-jupiter-params:$junitJupiterVersion"
    testRuntimeOnly "org.junit.jupiter:junit-jupiter-engine:$junitJupiterVersion"
    testImplementation "org.testcontainers:testcontainers:1.16.0"
    testImplementation "org.testcontainers:junit-jupiter:1.16.0"
}
```

Requires declaration of Jupiter and Testcontainers dependencies!

Creating a Container

```
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.junit.jupiter. Container;
import org.testcontainers.junit.jupiter. Testcontainers;

@Testcontainers
public class DatabaseIntegrationTest {

    @Container
    private static PostgreSQLContainer container =
        new PostgreSQLContainer("postgres:9.6.10-alpine")
            .withUsername("username")
            .withPassword("pwd")
            .withDatabaseName("todo");
}
```

Restarted vs. Shared Container

```
// Restarted for all test methods of class
@Container
private final PostgreSQLContainer container =
    new PostgreSQLContainer("postgres:9.6.10-alpine")
            .withUsername("username")
            .withPassword("pwd")
            .withDatabaseName("todo");

// Reused across all test methods of class
@Container
private static final PostgreSQLContainer container =
    new PostgreSQLContainer("postgres:9.6.10-alpine")
            .withUsername("username")
            .withPassword("pwd")
            .withDatabaseName("todo");
```

Instance field

Static field

Container Runtime Information

```
private CustomerRepository repository;

@BeforeEach
public void setUp() {
    String jdbcUrl = container.getJdbcUrl();
    int port = container.getFirstMappedPort();
    String username = container.getUsername();
    String password = container.getPassword();

    repository = new CustomerRepository(jdbcUrl, username,
password);
}
```

Accessing container runtime information

Injecting values into code under test

EXERCISE

Using
TestContainers for
a Java-based
Project with JUnit 5

Q & A

BREAK

Implementing Typical Integration Test Scenarios

Database Services, Multi-Services, Generic Containers

Testing Database Services

A common challenge for business applications

- Avoid using a local or shared, remote test database with state
- Testcontainers provides a wide range of database implementations as container images
- Seed data can be populated for each test scenario
- Managing the lifecycle of such a container is not as performant as H2

Adding the Dependency

Requires Testcontainers dependency and JDBC driver

build.gradle

```
dependencies {
    testImplementation 'org.testcontainers:mysql:1.16.0'
    runtimeOnly 'mysql:mysql-connector-java:8.0.26'
}
```

Database Container Object

API gives access to runtime connection information

```
import org.testcontainers.containers.MySQLContainer;

@Container
private final MySQLContainer container = new MySQLContainer();
```

```
String jdbcUrl = container.getJdbcUrl();
String username = container.getUsername();
String password = container.getPassword();
```

Creating Seed Data

Test cases require the database to be in a specific state

```
@Container
private final MySQLContainer container = new MySQLContainer()
    .withDatabaseName("accounting")
    .withInitScript("schema.sql");
```



```
CREATE TABLE customer(customer_id  INT NOT NULL AUTO_INCREMENT,
                      firstname  VARCHAR(100) NOT NULL,
                      lastname  VARCHAR(100) NOT NULL,
                      PRIMARY KEY (tutorial_id));
```

EXERCISE

Using a Database Module

Testing Multiple Services

Test scenarios may involved multiple services

- Testcontainers does not restrict your test to a single container
- Communication between containers can be established by setting up a network
- Docker Compose helps with defining multi-service setups in YAML

Instantiating Multiple Containers

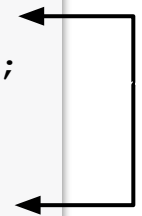
Simply create multiple instances

```
@Container
private final GenericContainer container1 =
    new GenericContainer("...");

@Container
private final GenericContainer container2 =
    new GenericContainer("...");
```

Shared Network Communication

Restricted to a single network per container

```
private final Network network = Network.newNetwork();

@Container
private final GenericContainer container1 =
    new GenericContainer("...").withNetwork(network);

@Container
private final GenericContainer container2 =
    new GenericContainer("...").withNetwork(network);
```



Can talk to each other

Docker Compose Container

Launches temporary Compose client

```
@Testcontainers
public class DockerComposeIntegrationTest {

   private final static File PROJECT_DIR = new File(System.getProperty("project.dir"));
   private final static String POSTGRES_SERVICE_NAME = "database_1";
   private final static int POSTGRES_SERVICE_PORT = 5432;

   @Container
   public static DockerComposeContainer environment = createComposeContainer();

   private static DockerComposeContainer createComposeContainer() {
       return new DockerComposeContainer(new File(PROJECT_DIR,
               "src/test/resources/compose-test.yml"))
               .withExposedService(POSTGRES_SERVICE_NAME, POSTGRES_SERVICE_PORT);
   }
}
```

Example Docker Compose File

Doesn't follow Compose YAML specification 100%

compose-test.yml

```
database:
  image: "postgres:9.6.10-alpine"
  environment:
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=postgres
    - POSTGRES_DB=todo
elasticsearch:
  image: "elasticsearch"
```

Compose allows defining one or more shared networks or use the default network

Accessing Runtime Information

Ambassador container makes port accessible to tests

```
private static String getPostgresServiceUrl() {
    String postgresHost =
        environment.getServiceHost(POSTGRES_SERVICE_NAME,
                                   POSTGRES_SERVICE_PORT);
    Integer postgresPort =
        environment.getServicePort(POSTGRES_SERVICE_NAME,
                                   POSTGRES_SERVICE_PORT);
    StringBuilder postgresServiceUrl =  new StringBuilder();
    postgresServiceUrl.append( "jdbc:postgresql://");
    postgresServiceUrl.append(postgresHost);
    postgresServiceUrl.append( ":");
    postgresServiceUrl.append(postgresPort);
    postgresServiceUrl.append( "/todo");
    return postgresServiceUrl.toString();
}
```

← IP address the container is listening to

← Exposed container port

Container Startup Timeout

Default to 60 secs per container but configurable

```
private final DockerComposeContainer environment =
    new DockerComposeContainer(new File(PROJECT_DIR,
                                        "src/test/resources/compose-test.yml"))
        .withExposedService(POSTGRES_SERVICE_NAME,
                            POSTGRES_SERVICE_PORT,
                            Wait.forListeningPort()
                                .withStartupTimeout(Duration.ofSeconds(120)));
```

Logic can also probe for HTTP endpoint or a log message

EXERCISE

Using the Docker Compose Module

Creating Generic Containers

Test scenarios may involved multiple services

- While there a specialized container implementations you may bring up any image in a container as a test fixture e.g. web servers, NoSQL database, or images built by other teams
- Testcontainers can build an image on-the-fly and use it
- Needs to provide the image at a minimum

Generic Container Usage

Use any container image you need

```
private static final DockerImageName IMAGE =

DockerImageName.parse("bmuschko/java-hello-world:1.0.0"));

private final GenericContainer container =
    new GenericContainer(IMAGE).withExposedPort(8080);
```

Building an Image On-The-Fly

For test cases from the end user's perspective

```
@Container
private final GenericContainer appContainer = createContainer();

private static GenericContainer createContainer() {
    return new GenericContainer(buildImageDockerfile())
            .withExposedPorts(8080)
            .waitingFor(Wait.forHttp("/actuator/health")
            .forStatusCode(200));
}

private static ImageFromDockerfile buildImageDockerfile() {
    return new ImageFromDockerfile()
            .withFileFromFile(ARCHIVE_NAME, new File(DISTRIBUTION_DIR, ARCHIVE_NAME))
            .withDockerfileFromBuilder(builder -> builder
                    .from("openjdk:jre-alpine")
                    .copy(ARCHIVE_NAME, "/app/" + ARCHIVE_NAME)
                    .entryPoint("java", "-jar", "/app/" + ARCHIVE_NAME)
                    .build());
}
```

EXERCISE

Using the Generic
Container Module

Q & A

BREAK

Going Further

Comparing Test Frameworks, Continuous Integration, Support for Other Languages

Test Framework Integration

TestContainers supports a wide range of options

- JUnit 4 - the legacy test framework which is still widely used
- JUnit 5 (Jupiter) - the current industry standard
- Spock framework - a powerful BDD test framework that requires the use Groovy for writing tests

JUnit 4 Implementation

No superclass, annotation on container object

```
public class DatabaseIntegrationTest {

    @Rule // or @ClassRule
    public final PostgreSQLContainer container =
        new PostgreSQLContainer("postgres:9.6.10-alpine");

    @Test
    public void testAccessDatabase() {
        // test case implementation
    }
}
```

Adding the Spock Dependency

Implemented as Spock extension

build.gradle

```
dependencies {
    testImplementation 'org.testcontainers:spock:1.16.0'
}
```

Spock Test Implementation

Extend Specification, mark with @Testcontainers

```
@Testcontainers
class DatabaseIntegrationTest extends Specification {

   @Shared
   PostgreSQLContainer container = new PostgreSQLContainer("postgres:9.6.10-alpine");

   def "can access database"() {
       given:
       // set up scenario

       when:
       // execute scenario

       then:
       // verify assertions
   }
}
```

Continuous Integration (CI)

Trigger an automated build for every commit

- Integrates changes into master/main branch
- Fast feedback by executing the build
- Use the same build tool as on a developer machine
- Standardizes on Bazel runtime version used

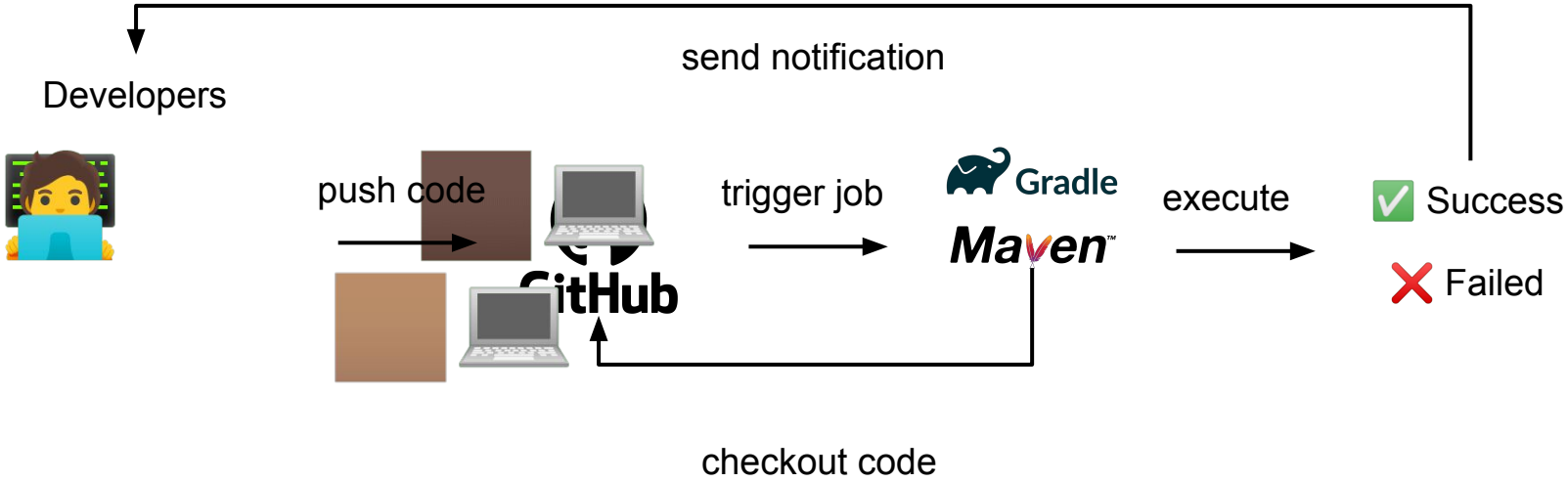
The CI Product GitHub Actions

Fully-integrated CI solution with GitHub repository

- Definition of build using a “configuration as code” approach
- Fast feedback by executing the build upon pushing a commit
- Use the same build tool and logic as on a developer machine

Basic Workflow

GitHub Actions reacts on an emitted repository event



Terminology

Essential for understanding a workflow definition

- **Event:** Repository activity that triggers a workflow
- **Job:** Set of steps that execute automation logic
- **Step:** Task that can run a command in a job
- **Action:** Reusable functionality provided by GitHub community

Typical Elements of Workflow File

Defines automation logic checked in GitHub repository

.github/workflows/build.yml

```yaml
name: Build and Release [Linux]
on: [push, pull_request]
jobs:
  build:
    name: Build
    runs-on: ubuntu-18.04
    steps:
      - name: Checkout
        uses: actions/checkout@v1
```

Event

Job

Step

Action

Using a Build Tool Action

Downloads and uses Gradle runtime

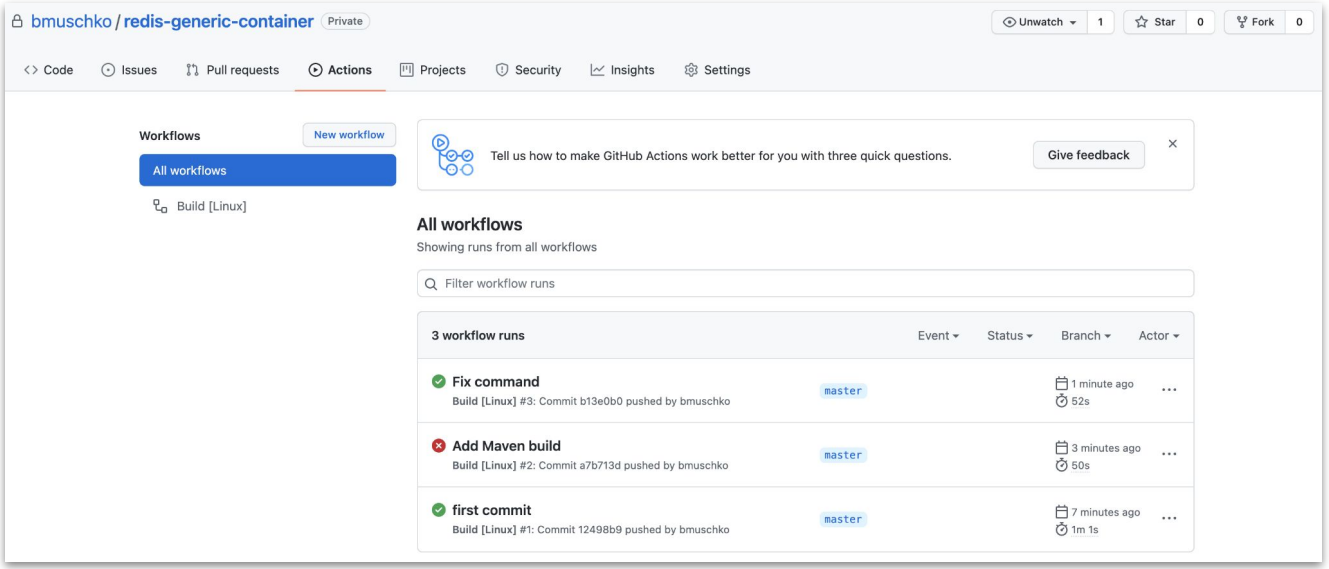
.github/workflows/build.yml

```yaml
steps:
- uses: actions/checkout@v2
- uses: actions/setup-java@v1
  with:
    java-version: 11
- uses: gradle/gradle-build-action@v2
  with:
    arguments: build
```

Build Gradle Action page

Actions in the Repository

Click on “Actions” tab at the top

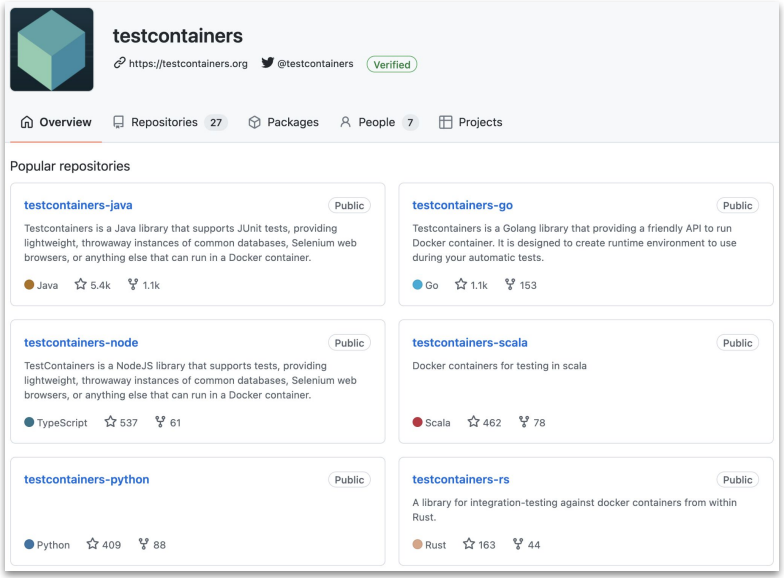


EXERCISE

Using
Testcontainers on
GitHub Actions

More Than Just Java

Support for other languages and ecosystems



Installing the Go Module

In your project, add library to Go Modules definition

```
$ go init github.com/bmuschko/redis-go
$ go get github.com/testcontainers/testcontainers-go
```

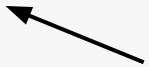
go.mod

```
module github.com/bmuschko/redis-go

go 1.17

require (
     github.com/testcontainers/testcontainers-go
v0.11.1
)
```

Using Testcontainers Go

Container lifecycle needs to controlled manually

```
func TestWithRedis(t *testing.T) {
    ctx := context.Background()
    req := testcontainers.ContainerRequest{
        Image:        "redis:latest",
        ExposedPorts: []string{"6379/tcp"},
        WaitingFor:   wait.ForLog("Ready to accept connections"),
    }
    redisC, err := testcontainers.GenericContainer(ctx, testcontainers.GenericContainerRequest{
        ContainerRequest: req,
        Started:          true,
    })
    if err != nil {
        t.Error(err)
    }
    defer redisC.Terminate(ctx)
}
```



Creation

Disposal

Q & A

Wrap Up

Summary and Lessons Learned

O'REILLY®

Thank you