TestContainers Singleton Pattern: Complete Guide

Overview

This guide explains how to solve TestContainers hanging issues using the Singleton pattern, organized according to the Diátaxis framework for clear, actionable documentation.

🎯 Tutorial: Implementing the Singleton Pattern

Step 1: Create the Base Integration Test Class

java

```java
public abstract class AbstractIntegrationTest {
    private static final PostgreSQLContainer<?> POSTGRES_CONTAINER;

    static {
        POSTGRES_CONTAINER = new PostgreSQLContainer<>("postgres:15-alpine")
                .withDatabaseName("testdb")
                .withUsername("test")
                .withPassword("test")
                .withReuse(true);

        POSTGRES_CONTAINER.start();

        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            if (POSTGRES_CONTAINER.isRunning()) {
                POSTGRES_CONTAINER.stop();
            }
        }));
    }

    @DynamicPropertySource
    static void configureProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", POSTGRES_CONTAINER::getJdbcUrl);
        registry.add("spring.datasource.username", POSTGRES_CONTAINER::getUsername);
        registry.add("spring.datasource.password", POSTGRES_CONTAINER::getPassword);
    }
}
```

Step 2: Extend in Your Test Classes

```java
@SpringBootTest
@Transactional
class DeviceRepoIntegrationTest extends AbstractIntegrationTest {

    @Autowired
    private DeviceRepo deviceRepo;

    @Test
    void testYourMethod() {
        // Your test logic here
        // The container is automatically available
    }
}
```

Step 3: Run Your Tests

```bash
./gradlew test
# Tests will run without hanging!
```

📖 How-To Guides

How to Optimize Container Performance

Add these configurations to your container setup:

```java
POSTGRES_CONTAINER = new PostgreSQLContainer<>("postgres:15-alpine")
    .withTmpFs(Map.of("/var/lib/postgresql/data", "rw")) // Use memory filesystem
    .withCommand("postgres",
        "-c", "fsync=off",              // Disable disk sync
        "-c", "synchronous_commit=off", // Async commits
        "-c", "max_connections=50")     // Limit connections
    .withReuse(true);
```

How to Configure Connection Pool Settings

Add these properties via `@DynamicPropertySource`:

```java
registry.add("spring.datasource.hikari.maximum-pool-size", () -> "2");
registry.add("spring.datasource.hikari.connection-timeout", () -> "5000");
registry.add("spring.datasource.hikari.idle-timeout", () -> "30000");
```

How to Debug Container Issues

Enable detailed logging temporarily:

```java
registry.add("logging.level.org.testcontainers", () -> "DEBUG");
registry.add("logging.level.com.github.dockerjava", () -> "DEBUG");
```

How to Handle Multiple Database Types

Create specific abstract classes:

```java
public abstract class PostgreSQLIntegrationTest extends AbstractIntegrationTest {
    // PostgreSQL-specific configuration
}

public abstract class MySQLIntegrationTest {
    private static final MySQLContainer<?> MYSQL_CONTAINER;
    // MySQL-specific singleton setup
}
```

🔍 Explanation: Why the Singleton Pattern Works

The Problem with Traditional Approaches

Traditional TestContainers Setup:

```java
```

```
@Testcontainers
public class MyTest {
    @Container
    static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>();

    @AfterAll
    static void cleanup() {
        postgres.stop(); // ❌ Causes hanging
    }
}
```

Issues:

- Multiple start/stop cycles across test classes
- Race conditions between manual cleanup and Ryuk
- Connection pool drainage conflicts
- Docker resource cleanup timing issues

How Singleton Pattern Solves This

Container Lifecycle:

```
JVM Startup → Static Block → Container Starts (ONCE)
    ↓
Test Class A → Uses Container → Tests Run
Test Class B → Uses Container → Tests Run
Test Class C → Uses Container → Tests Run
    ↓
JVM Shutdown → Shutdown Hook → Container Stops (ONCE)
```

Key Benefits:

1. **Single Lifecycle Management**
   - Container starts once when class loads
   - Container stops once when JVM shuts down
   - Eliminates multiple start/stop cycles
2. **Proper Cleanup Timing**
   - Shutdown hook runs at JVM termination
   - No conflicts with TestContainers' Ryuk cleanup

- Natural integration with Spring context shutdown

3. **Resource Efficiency**
   - One container serves all test classes
   - Faster test execution (no startup overhead)
   - Reduced Docker resource churn
4. **Connection Pool Harmony**
   - Connection pools close naturally with Spring context
   - No abrupt connection termination
   - Reduced connection cleanup conflicts

Technical Deep Dive

Static Initialization Block:

```java
static {
    POSTGRES_CONTAINER = new PostgreSQLContainer<>(...);
    POSTGRES_CONTAINER.start(); // Happens once per JVM
}
```

- Executes when class is first loaded
- Guarantees single container instance
- Thread-safe initialization

Shutdown Hook:

```java
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    if (POSTGRES_CONTAINER.isRunning()) {
        POSTGRES_CONTAINER.stop();
    }
}));
```

- Registers cleanup with JVM shutdown sequence
- Executes after all test classes complete
- Prevents resource leaks

📚 Reference

Configuration Options

Container Configuration

Option	Purpose	Recommended Value
`withReuse(true)`	Enable container reuse	`true`
`withTmpFs()`	Use memory filesystem	`/var/lib/postgresql/data`
`withCommand()`	PostgreSQL optimization	See performance settings

PostgreSQL Performance Settings

Setting	Purpose	Test Value
`max_connections`	Limit connections	`50`
`fsync`	Disable disk sync	`off`
`synchronous_commit`	Async commits	`off`
`shared_buffers`	Memory allocation	`128MB`

HikariCP Settings

Property	Purpose	Recommended
`maximum-pool-size`	Max connections	`2`
`connection-timeout`	Connection wait time	`5000ms`
`idle-timeout`	Idle connection timeout	`30000ms`
`max-lifetime`	Connection max age	`60000ms`

Common Patterns

Basic Singleton Pattern

```java
```

```
private static final PostgreSQLContainer<?> POSTGRES_CONTAINER;

static {
    POSTGRES_CONTAINER = new PostgreSQLContainer<>("postgres:15-alpine");
    POSTGRES_CONTAINER.start();

    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        if (POSTGRES_CONTAINER.isRunning()) {
            POSTGRES_CONTAINER.stop();
        }
    }));
}
```

Optimized Singleton Pattern

java

```
private static final PostgreSQLContainer<?> POSTGRES_CONTAINER;

static {
    POSTGRES_CONTAINER = new PostgreSQLContainer<>("postgres:15-alpine")
        .withReuse(true)
        .withTmpFs(Map.of("/var/lib/postgresql/data", "rw"))
        .withCommand("postgres", "-c", "fsync=off", "-c", "synchronous_commit=off");

    POSTGRES_CONTAINER.start();

    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        try {
            if (POSTGRES_CONTAINER.isRunning()) {
                POSTGRES_CONTAINER.stop();
            }
        } catch (Exception e) {
            System.err.println("Error stopping container: " + e.getMessage());
        }
    }));
}
```

Troubleshooting

Container Not Starting

- Check Docker daemon is running

- Verify image availability: `docker pull postgres:15-alpine`
- Check port conflicts

Still Hanging After Implementation

- Disable Ryuk: `TESTCONTAINERS_RYUK_DISABLED=true`
- Check Gradle daemon: `./gradlew --stop`
- Verify shutdown hook registration

Performance Issues

- Enable container reuse: `withReuse(true)`
- Use tmpfs for data directory
- Optimize PostgreSQL settings
- Reduce connection pool size

Best Practices

1. **Always use shutdown hooks** instead of `@AfterAll`
2. **Enable container reuse** for faster tests
3. **Minimize connection pool size** for integration tests
4. **Use memory filesystem** (tmpfs) for better performance
5. **Disable unnecessary logging** to reduce overhead
6. **Configure timeouts** to prevent indefinite hanging