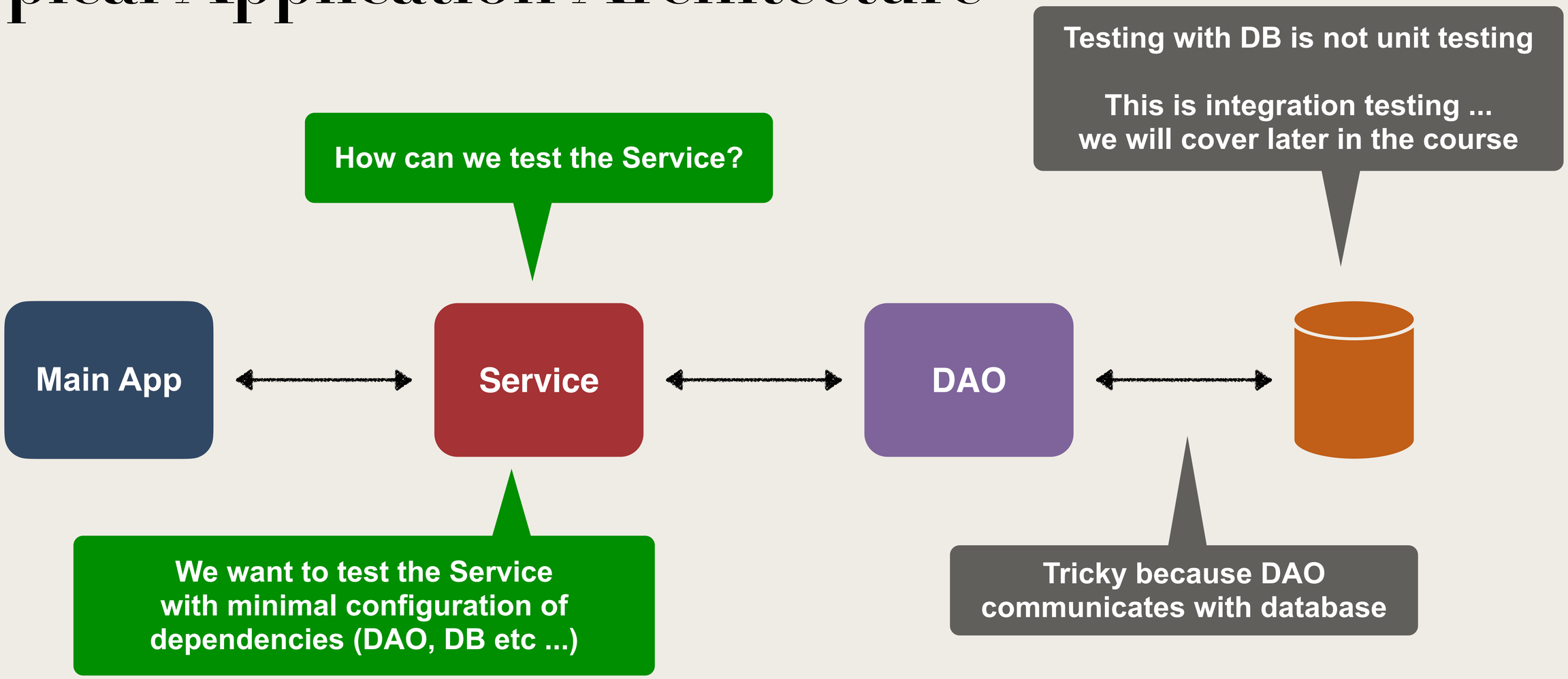
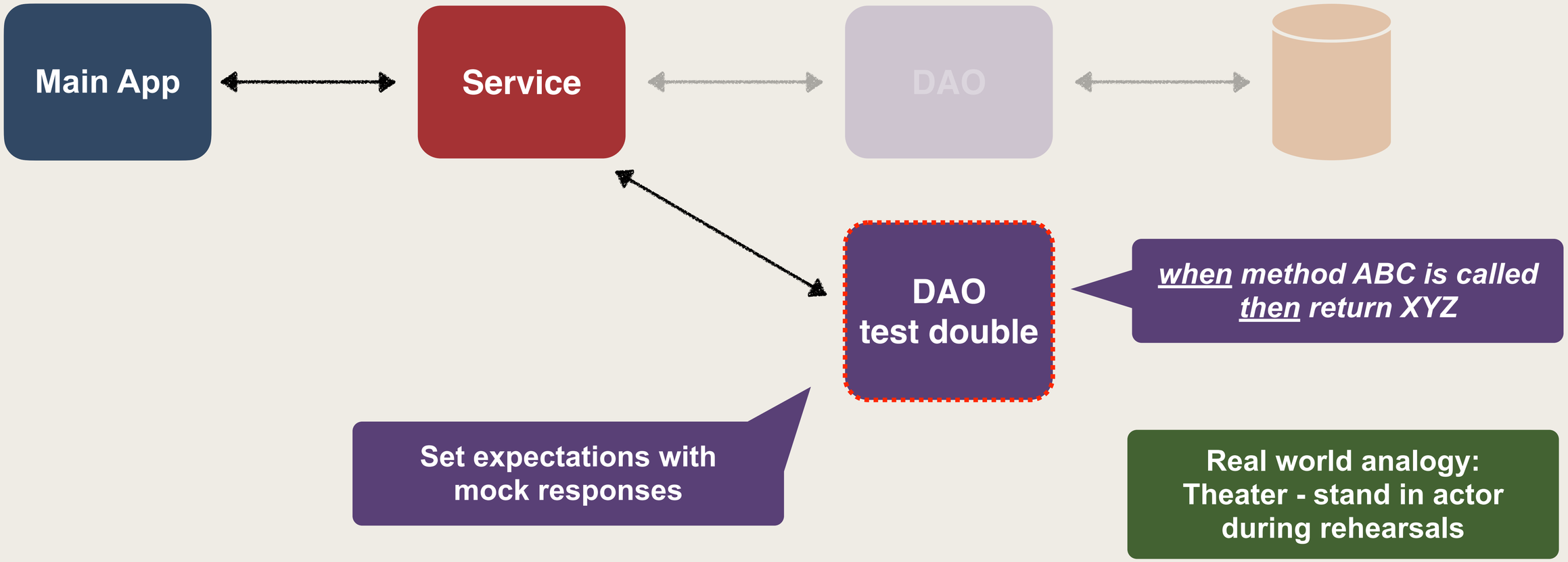
luv2code

Mocks with Mockito and Spring Boot

Typical Application Architecture



Using a Test Double



The technique of using test doubles is known as "mocking"

Benefits of Mocking

- Allows us to test a given class in isolation
- Test interaction between given class and its dependencies
- Minimizes configuration / availability of dependencies
- For example DAO, DB, REST API etc
  - We can mock the DAO to give a response
  - We can mock a REST API to give a response

Real world analogy:
Theater - stand in actor during rehearsals

Mocking Frameworks

- The Java ecosystem includes a number of Mocking frameworks
- The Mocking frameworks provide following features:
  - Minimize hand-coding of mocks ... leverage annotations
  - Set expectations for mock responses
  - Verify the calls to methods including the number of calls
  - Programmatic support for throwing exceptions

Mocking Frameworks

Name	Website
Mockito	`site.mockito.org`
EasyMock	`www.easymock.org`
JMockit	`jmockit.github.io`
...	

We will use Mockito since
Spring Boot has built-in support for Mockito

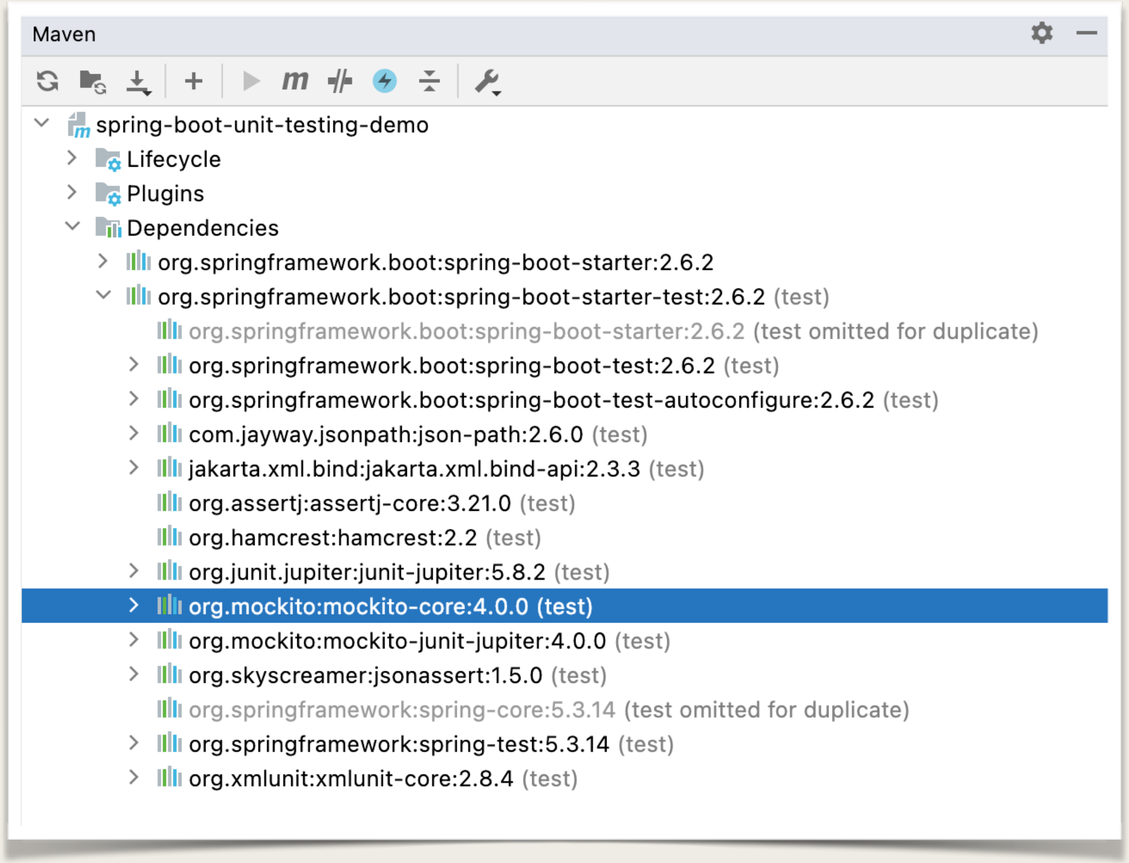
Spring Boot Starter - Transitive Dependency for Mockito

pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

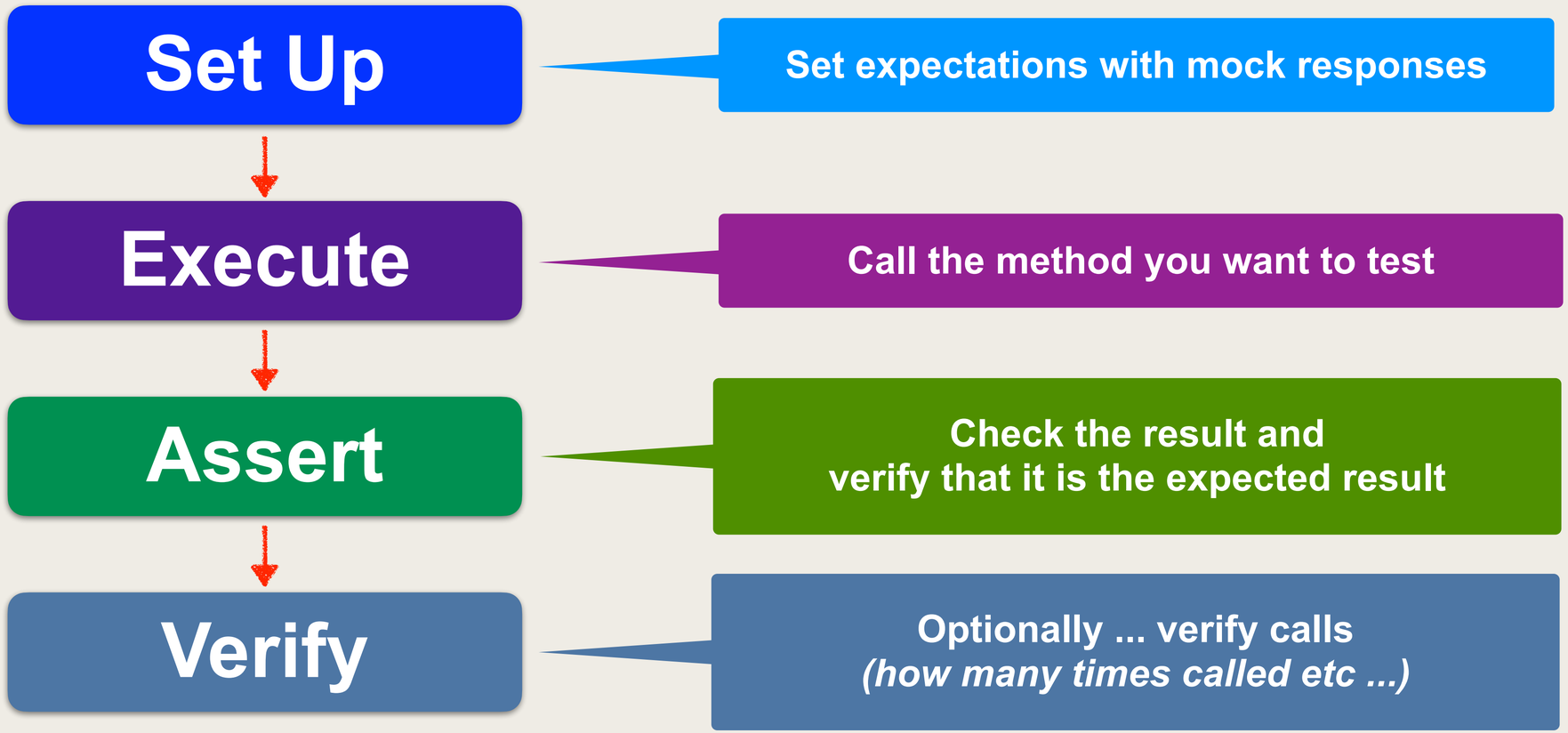
Starter includes a transitive dependency on Mockito

We get it for free :-)

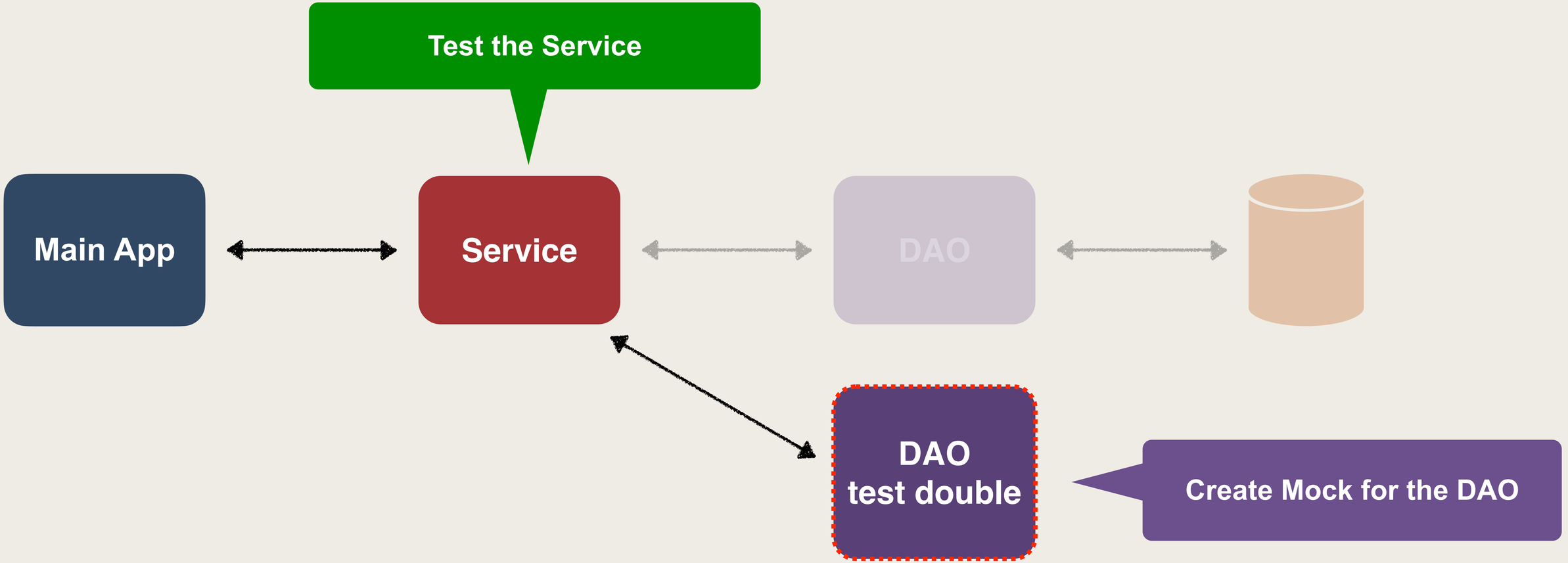


Unit testing with Mocks

- Unit tests with Mocks have the following structure



Testing Plan

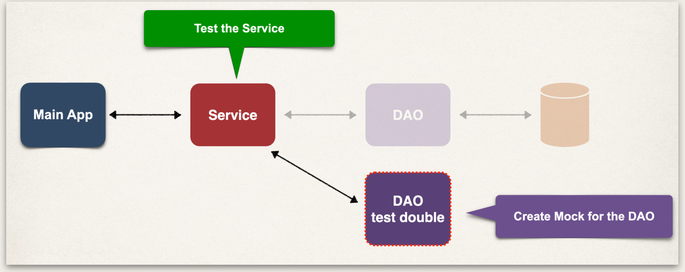


Review code for Service and DAO

ApplicationService.java

```
public class ApplicationService {

  @Autowired
  private ApplicationDao applicationDao;

  public double addGradeResultsForSingleClass(List<Double> grades) { ... }

  public double findGradePointAverage (List<Double> grades ) { ... }

  public Object checkNull(Object obj) { ... }

}
```

Dependency



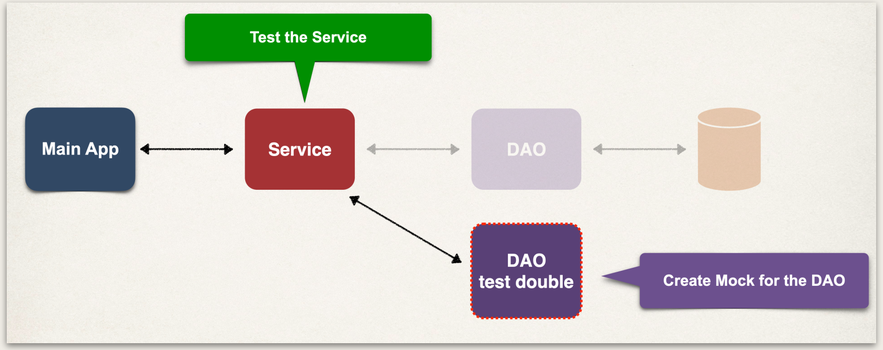
ApplicationDao.java

```
public class ApplicationDao {

  public double addGradeResultsForSingleClass(List<Double> grades) { ... }

  public double findGradePointAverage (List<Double> grades ) { ... }

  public Object checkNull(Object obj) { ... }

}
```

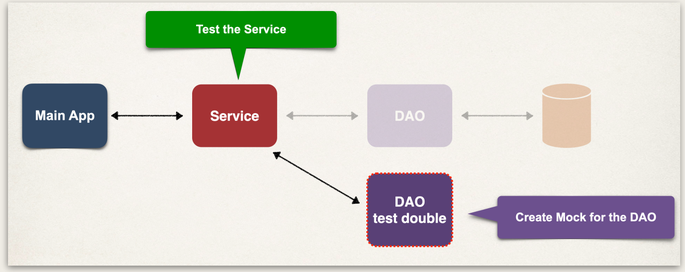
Development Process

Step-By-Step

1. Create Mock for DAO
2. Inject mock into Service
3. Set up expectations
4. Call method under test and assert results
5. Verify method calls



Step 1: Create Mock for the DAO

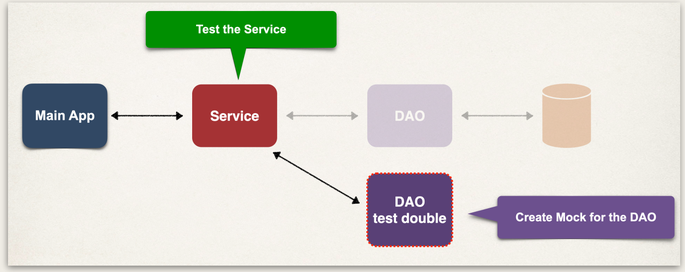


MockAnnotationTest.java

```
import org.mockito.Mock;
...

@SpringBootTest(classes=MvcTestingExampleApplication.class)
public class MockAnnotationTest {

  @Mock
  private ApplicationDao applicationDao;


  ...

}
```

Create Mock for the DAO

Step 2: Inject mock into Service



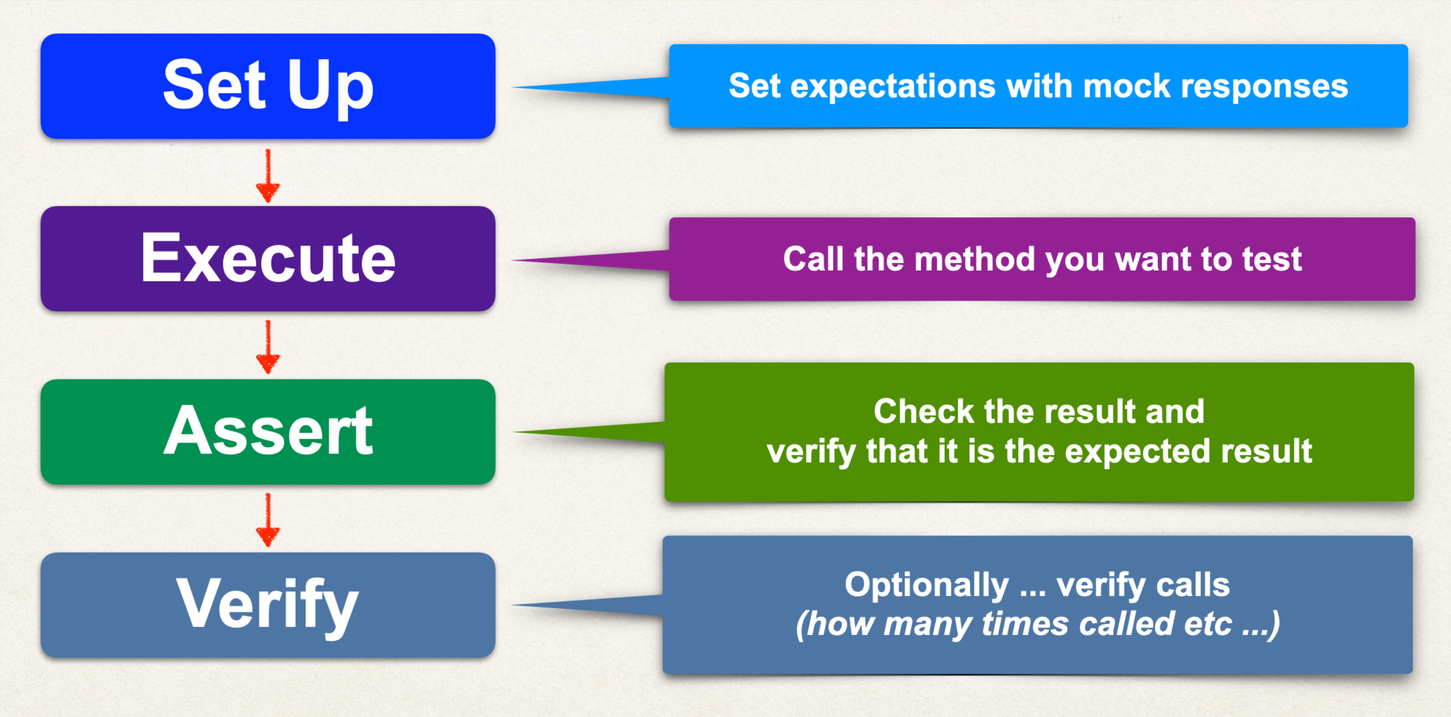
MockAnnotationTest.java

```
import org.mockito.Mock;
import org.mockito.InjectMocks;
...

@SpringBootTest(classes=MvcTestingExampleApplication.class)
public class MockAnnotationTest {

  @Mock
  private ApplicationDao applicationDao;

  @InjectMocks
  private ApplicationService applicationService;

  ...

}
```

Inject mock dependencies

Note: Will only inject dependencies annotated with @Mock or @Spy

Step 3: Set up expectations



Step 3: Set up expectations

when ***method***
doSomeWork(...)
is called
then return **"I am finished"**

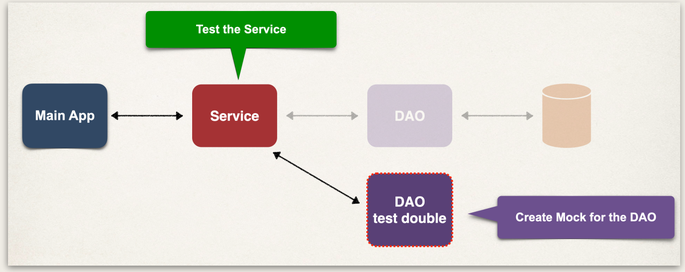
```
import static org.mockito.Mockito.when;
...

    String aResponse = "I am finished";

    when( doSomeWork() ).thenReturn( aResponse );
```

response

method

Real world analogy:
Theater - just read the script

Step 3: Set up expectations



MockAnnotationTest.java

```
import static org.mockito.Mockito.when;
import static org.junit.jupiter.api.Assertions.assertEquals;
...

@SpringBootTest(classes=MvcTestingExampleApplication.class)
public class MockAnnotationTest {

  @Mock
  private ApplicationDao applicationDao;

  @InjectMocks
  private ApplicationService applicationService;

  @Autowired
  private CollegeStudent studentOne;

  @Autowired
  private StudentGrades studentGrades;


  @DisplayName("When & Verify")
  @Test
  public void assertEqualsTestAddGrades() {
      when(applicationDao.addGradeResultsForSingleClass(
                              studentGrades.getMathGradeResults())).thenReturn(100.0);

      ...
  }
}
```

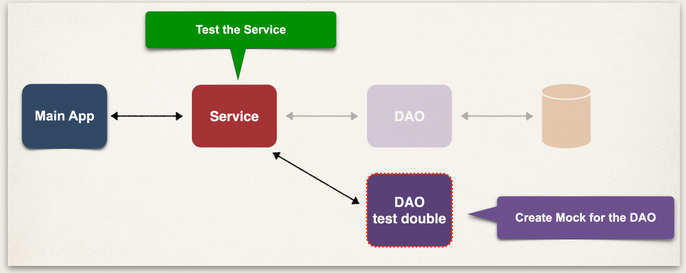
Set up expectations for mock response

***when* method addGradeResultsForSingleClass(...) *is called* *then return* 100.0**

Step 4: Call method under test and assert results

MockAnnotationTest.java

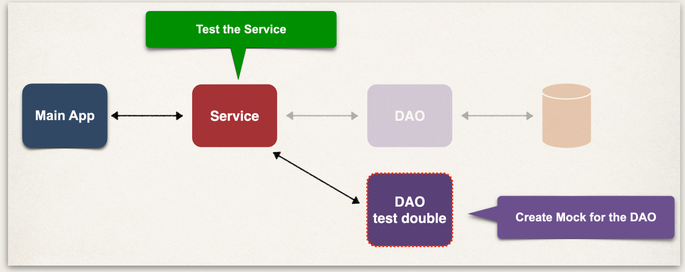
```
import static org.mockito.Mockito.when;
import static org.junit.jupiter.api.Assertions.assertEquals;
...

@SpringBootTest(classes= MvcTestingExampleApplication.class)
public class MockAnnotationTest {

  @Mock
  private ApplicationDao applicationDao;

  @InjectMocks
  private ApplicationService applicationService;

  @Autowired
  private CollegeStudent studentOne;

  @Autowired
  private StudentGrades studentGrades;


  @DisplayName("When & Verify")
  @Test
  public void assertEqualsTestAddGrades() {
      when(applicationDao.addGradeResultsForSingleClass(
                              studentGrades.getMathGradeResults())).thenReturn(100.0);

      assertEquals(100.0, applicationService.addGradeResultsForSingleClass(
                              studentOne.getStudentGrades().getMathGradeResults()));

  }
}
```



Assert results

The service uses the DAO ...
that has been set up to return 100.0

Step 5: Verify method calls

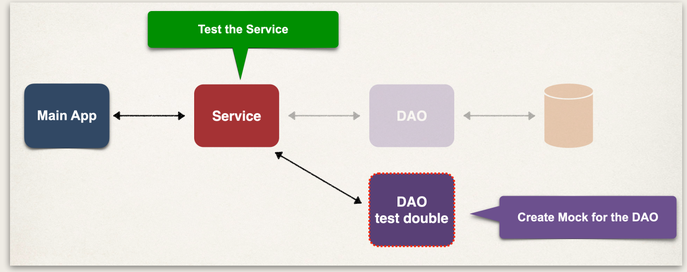


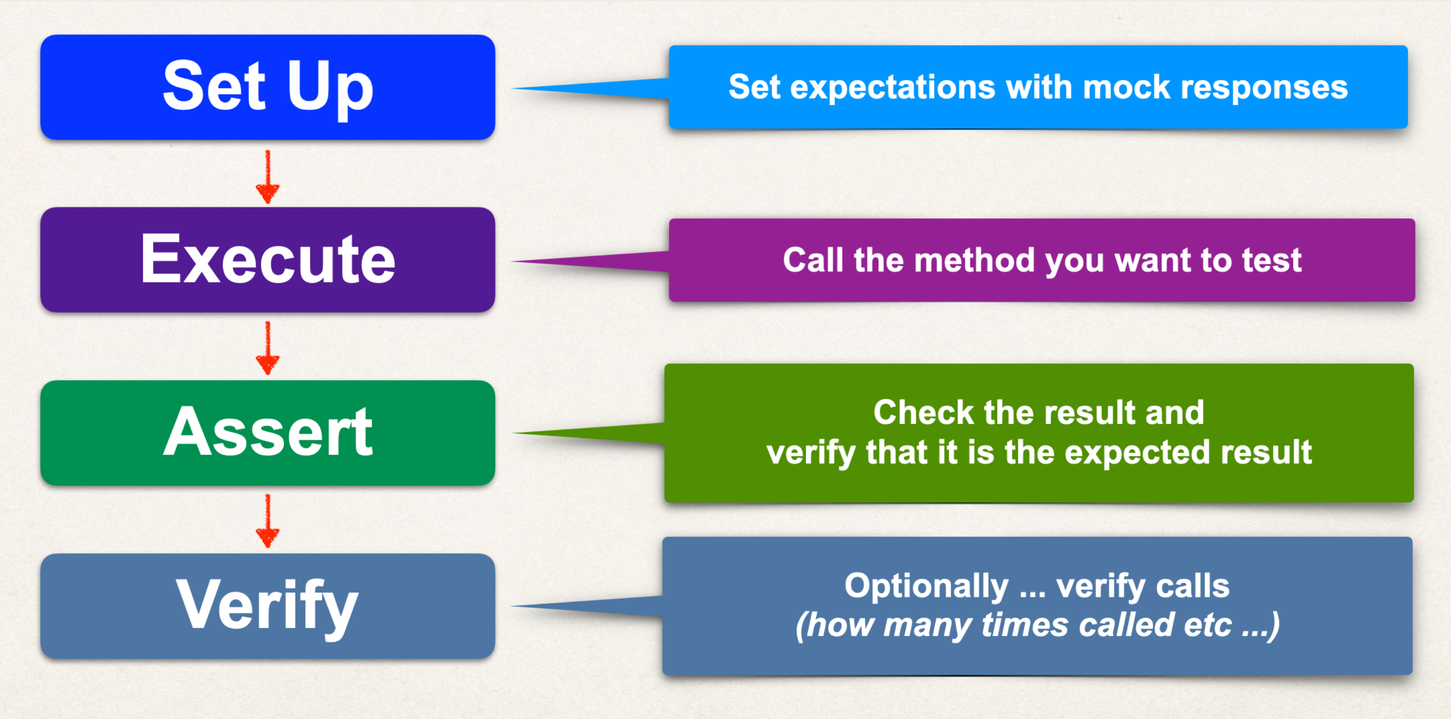
MockAnnotationTest.java

```
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.times;
...

@SpringBootTest(classes= MvcTestingExampleApplication.class)
public class MockAnnotationTest {

  @Mock
  private ApplicationDao applicationDao;

  @InjectMocks
  private ApplicationService applicationService;

  ...

  @DisplayName("When & Verify")
  @Test
  public void assertEqualsTestAddGrades() {
      when(applicationDao.addGradeResultsForSingleClass(
                              studentGrades.getMathGradeResults())).thenReturn(100.0);

      assertEquals(100.0, applicationService.addGradeResultsForSingleClass(
                              studentOne.getStudentGrades().getMathGradeResults()));

     verify(applicationDao,
               times(1)).addGradeResultsForSingleClass(studentGrades.getMathGradeResults());

  }
}
```

Verify the DAO method was called 1 time

Recap





Mockito Resources

- Additional features
  - Stubs, spies
  - Argument matchers, Answers
  - ...

site.mockito.org

luv2code

Spring Boot @MockBean

MockAnnotationTest.java

```
import org.mockito.Mock;
import org.mockito.InjectMocks;
...

@SpringBootTest(classes=MvcTestingExampleApplication.class)
public class MockAnnotationTest {

  @Mock
  private ApplicationDao applicationDao;

  @InjectMocks
  private ApplicationService applicationService;

  ...

}
```

Inject mock dependencies

Note: Will only inject dependencies annotated with @Mock or @Spy

Spring Boot @MockBean

- Instead of using Mockito: `@Mock` and `@InjectMocks`
- Use Spring Boot support: `@MockBean` and `@Autowired`

- `@MockBean`
  - includes Mockito `@Mock` functionality
  - also adds mock bean to Spring ApplicationContext
    - if existing bean is there, the mock bean will replace it
  - thus making the mock bean available for injection with `@Autowired`

Use Spring Boot @MockBean
when you need to inject mocks
AND
inject regular beans from app context

BEFORE

MockAnnotationTest.java

```
import org.mockito.Mock;
import org.mockito.InjectMocks;
...

@SpringBootTest(classes=MvcTestingExampleApplication.class)
public class MockAnnotationTest {

  @Mock
  private ApplicationDao applicationDao;

  @InjectMocks
  private ApplicationService applicationService;

  ...

}
```

AFTER

MockAnnotationTest.java

```
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.beans.factory.annotation.Autowired;
...

@SpringBootTest(classes=MvcTestingExampleApplication.class)
public class MockAnnotationTest {

  @MockBean
  private ApplicationDao applicationDao;

  @Autowired
  private ApplicationService applicationService;

  ...

}
```

Create Mock for the DAO

Inject dependencies

No longer a limitation ...
can inject any dependency ...
mock or regular bean

Throwing Exceptions with Mocks

Throwing Exceptions

- May need to configure the mock to throw an exception
- Possible use case
  - Testing how the code handles exceptions

Refresher

MockAnnotationTest.java

```
...

@SpringBootTest(classes=MvcTestingExampleApplication.class)
public class MockAnnotationTest {

 @MockBean
 private ApplicationDao applicationDao;

 @Autowired
 private ApplicationService applicationService;

 ...

}
```

We will mock the DAO to throw exceptions

Throw Exception

MockAnnotationTest.java

```
@DisplayName("Thrown an Exception")
@Test
public void throwAnException() {
    CollegeStudent nullStudent = (CollegeStudent) context.getBean("collegeStudent");

    when(applicationDao.checkNull(nullStudent))
            .thenThrow(new RuntimeException());

    assertThrows(RuntimeException.class, () -> {
        applicationService.checkNull(nullStudent);
    });
}
```

When the checkNull(...) method is called ... thenThrow an exception

Assert that the exception was thrown

Throw Exception: Consecutive calls

MockAnnotationTest.java

```
@DisplayName("Multiple Stubbing")
@Test
public void stubbingConsecutiveCalls() {
    CollegeStudent nullStudent = (CollegeStudent) context.getBean("collegeStudent");

    when(applicationDao.checkNull(nullStudent))
            .thenThrow(new RuntimeException())
            .thenReturn("Do not throw exception second time");

    assertThrows(RuntimeException.class, () -> {
        applicationService.checkNull(nullStudent);
    });

    assertEquals("Do not throw exception second time", applicationService.checkNull(nullStudent));

}
```

First call ... throw exception

Consecutive calls ... do NOT throw exception
Just return a string

First call

Second call