Testing Spring Boot Applications

1. The Spring Boot test ecosystem
2. Writing and running tests on beans
3. Mocking beans
4. Additional Spring Boot test techniques

Demo project: **`demo-testing`**

1. The Spring Boot Test Ecosystem

- Overview
- Spring Boot test dependency
- Defining test cases in Spring Boot
- Understanding `@SpringBootTest`
- Specifying Java config classes for tests
- Specifying properties for tests
- Specifying a web environment for tests

Demo package: **`demo.testing`**

Overview

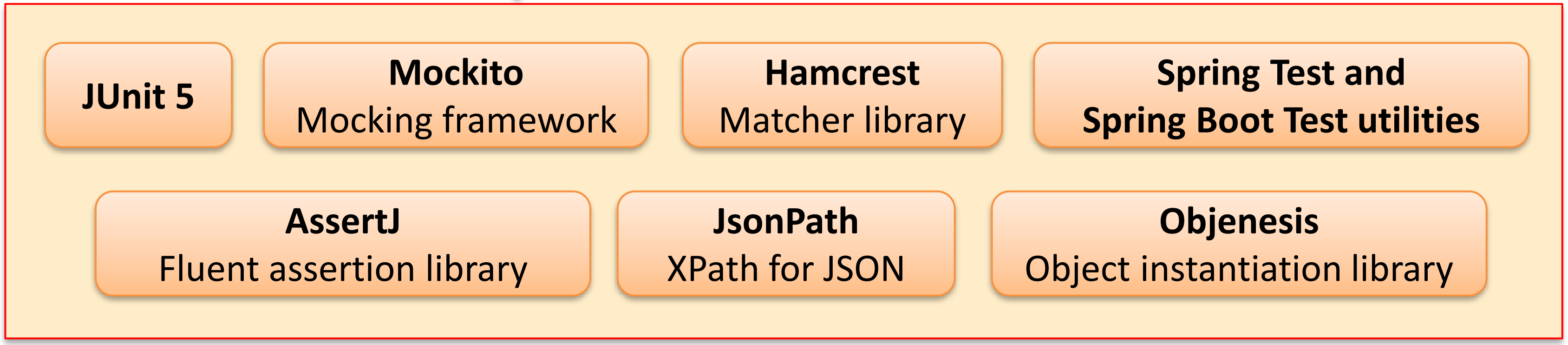
- Spring Boot makes testing easy, in various ways…
- Spring Boot auto-configuration automatically sucks in common test libraries
- Spring Boot automatically makes components available for autowiring into your test cases
- Spring Boot automatically loads application properties, so you can test components with realistic settings

Spring Boot Test Dependency

- When you create a Spring Boot app with Spring Initializr, it has the `spring-boot-starter-test` dependency

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

pom.xml



JUnit 5
Mockito Mocking framework
Hamcrest Matcher library
Spring Test and Spring Boot Test utilities
AssertJ Fluent assertion library
JsonPath XPath for JSON
Objenesis Object instantiation library

olsen software

Defining Test Cases in Spring Boot

- Spring Initializr also generates a simple JUnit test case
  - See the `src/test/java` folder in the demo project

```
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class ApplicationTests {

    @Test
    void contextLoads() {
    }
    …
}
```

ApplicationTests.java

- We discuss the details on the following slides…

Understanding @SpringBootTest

```
@SpringBootTest
class ApplicationTests {
    …
}
```

- `@SpringBootTest` automatically loads Java config classes, which presumably define the beans you want to test
  - First it loads inner classes annotated with `@Configuration`
  - If none found, it loads your `@SpringBootApplication` class

- `@SpringBootTest` also loads `application.properties`
  - So the beans are initialized properly when you test them

Specifying Java Config Classes for Tests

- `@SpringBootTest` **has a** `classes` **attribute**
  - Specifies particular Java config classes you want to load
  - Enables you to control which beans are created for your tests

```
@SpringBootTest(classes={MyJavaConfig1.class, MyJavaConfig2.class})
public class ApplicationTests {
      …
}
```

Specifying Properties for Tests

- `@SpringBootTest` **has a** `properties` **attribute**
  - Specifies additional properties you want to use in your tests
  - You specify an array of key=value strings

```
@SpringBootTest(properties={"prop1=value1", "prop2=value2"})
public class ApplicationTests {
    …
}
```

Specifying a Web Environment for Tests

- `@SpringBootTest` has a `webEnvironment` attribute
  - Enables you to configure a web environment for your tests

- To use a mock servlet environment:

```
@SpringBootTest(webEnvironment=SpringBootTest.WebEnvironment.MOCK)
```

- To use a real server on the port defined by `server.port`:

```
@SpringBootTest(webEnvironment=SpringBootTest.WebEnvironment.DEFINED_PORT)
```

- To use a real web server on a random port number:

```
@SpringBootTest(webEnvironment=SpringBootTest.WebEnvironment.RANDOM_PORT)
```

2. Writing and Running Tests on Beans

- Defining a bean to test
- Writing a test for a bean
- Running tests

Defining a Bean to Test

- Here's a simple bean to test

```
@Component
public class BankAccountBean {

    private String name;
    private int balance = 0;

    public void deposit(int amount) {
        balance += amount;
    }

    public void withdraw(int amount) {
        if (amount > balance)
           throw new IllegalArgumentException("Insufficient funds");
        balance -= amount;
    }

    // Plus getters, setters, toString(), etc.
}
```

BankAccountBean.java

olsen software

Writing a Test for a Bean

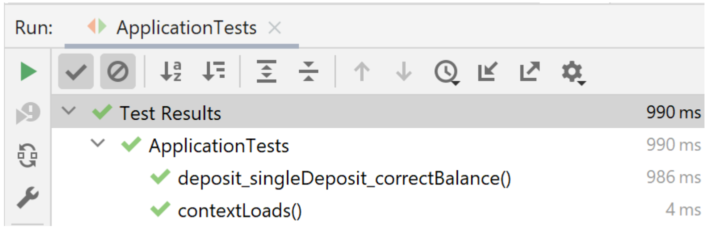
- Here's how we can test the bean
  - Spring loads the application context, as previously discussed
  - So we can autowire the bean into our test case, and then test it

```
import static org.junit.jupiter.api.Assertions.assertEquals;

@SpringBootTest
public class ApplicationTests {

    @Autowired
    BankAccountBean fixture;

    @Test
    public void deposit_singleDeposit_correctBalance() {
        fixture.deposit(100);
        assertEquals(100, fixture.getBalance());
    }
}
```

ApplicationTests.java

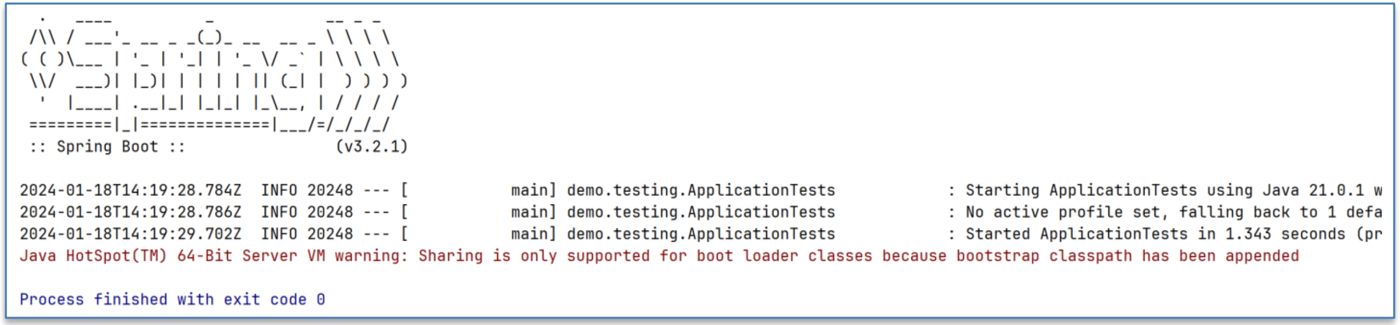
Running Tests

- Run tests as normal
  - See if the tests pass or fail



```
Run:   ApplicationTests
Test Results                                  990 ms
  ApplicationTests                            990 ms
    deposit_singleDeposit_correctBalance()    986 ms
    contextLoads()                              4 ms
```

- Also note the info displayed in the console
  - Indicates the Spring application context has been loaded



```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.2.1)

2024-01-18T14:19:28.784Z  INFO 20248 --- [           main] demo.testing.ApplicationTests            : Starting ApplicationTests using Java 21.0.1 w
2024-01-18T14:19:28.786Z  INFO 20248 --- [           main] demo.testing.ApplicationTests            : No active profile set, falling back to 1 defa
2024-01-18T14:19:29.702Z  INFO 20248 --- [           main] demo.testing.ApplicationTests            : Started ApplicationTests in 1.343 seconds (pr
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended

Process finished with exit code 0
```

olsen software

3. Mocking Beans

- Overview
- Java mocking frameworks
- Example bean to test
- Testing the bean using Mockito mocks

Overview

- Object-oriented systems involve lots of interacting objects
    - Unit testing focuses on the behaviour of an isolated object

- We can use a mocking framework to create a "mock" of other objects that we use
    - Specify what methods you expect to be called on a mock object
    - Specify what you want the methods to return

Java Mocking Frameworks

- There are various Java mocking frameworks available:
  - Mockito
  - jMock
  - EasyMock
  - Mock Objects
  - etc…
- The `spring-boot-starter-test` dependency automatically includes the Mockito library
  - To use an alternative mocking framework, add the appropriate dependency to your pom file

Example Bean to Test

- Here's an example bean that we're going to test
  - Note that it has an autowired `BARepository` dependency

```
@Component
public class BAServiceBean {

    private BARepository repo;

    @Autowired
    public BAServiceBean(BARepository repo) {
        this.repo = repo;
    }

    public void depositIntoAccount(int id, int amount) {
        BankAccountBean acc = repo.getById(id);
        acc.deposit(amount);
        repo.update(id, acc);
    }
}
```

BAServiceBean.java

```
public interface BARepository {
    …
}
```

olsen software

Testing the Bean using Mockito Mocks (1 of 2)

- Spring Boot has a `@MockBean` annotation
  - Tells Mockito to create a mock bean in the application context

```
@SpringBootTest
public class BAServiceBeanTests {

    @MockBean
    private BARepository mockRepo;

    @Autowired
    BAServiceBean service;

    …
}
```

BAServiceBeanTest.java

Spring Boot will create a mock instance of `BARepository` in the application context

Spring Boot will inject the `BARepository` mock bean into the `BAServiceBean` bean

Testing the Bean using Mockito Mocks (2 of 2)

- You can now write tests as follows:

```
@SpringBootTest
public class BAServiceBeanTests {

    …

    @Test
    public void testDeposit() {
        BankAccountBean acc = new BankAccountBean();
        when(mockRepo.getById(anyInt())).thenReturn(acc);

        service.depositIntoAccount(1234, 100);
        assertEquals(acc.getBalance(), 100);

        verify(mockRepo).getById(eq(1234));
        verify(mockRepo).update(eq(1234), refEq(acc));
    }
}
```

BAServiceBeanTest.java

Specify return value for mocked methods

Verify mocked methods were called as expected

4. Additional Spring Boot Test Techniques

- Testing Spring Data repositories
- Testing REST controllers

Testing Spring Data Repositories (1 of 2)

- Earlier in the course we discussed Spring Data repositories
  - Define an interface that extends `CrudRepository`
  - Define query methods, which Spring automatically implements

```
public interface EmployeeRepository extends CrudRepository<Employee, Long> {

    List<Employee> findsByRegion(String region);

    @Query("select e from Employee e where e.dosh >= ?1 and e.dosh <= ?2")
    List<Employee> findInSalaryRange(double from, double to);

    Page<Employee> findByDoshGreaterThan(double sal, Pageable pageable);
}
```

- How can you test these repositories?
  - See next slide…

Testing Spring Data Repositories (2 of 2)

- Spring Boot makes it easy to test Spring Data repositories
  - Define a test class and annotate with `@DataJpaTest`
  - Use a `TestEntityManager` to prepare database state

```
@DataJpaTest  // Configures in-mem db, and does JPA-related config only.
public class EmployeeRepositoryTest {

    @Autowired
    private TestEntityManager em;  // Has some additional test-related APIs.

    @Autowired
    private EmployeeRepository repository;

    @Test
    public void testFindByRegion() {
        em.persist(new Employee(-1, "John Smith", 25000, "London"));
        em.persist(new Employee(-1, "Jane Evans", 30000, "Dublin"));
        List<Employee> emps = repository.findByRegion("London");
        assertEquals(1, emps.size());
    }
}
```

olsen software

Testing REST Controllers

- Spring Boot makes it easy to test REST controllers
  - In `@SpringBootTest`, set the `webEnvironment` property
  - Inject a `TestRestTemplate`, a test-friendly version of `RestTemplate` that doesn't throw exceptions for server errors

```
@SpringBootTest(webEnvironment=SpringBootTest.WebEnvironment.RANDOM_PORT)
public class SomeRestControllerTests {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void testGetAllProducts() {

        ResponseEntity<List<Product>> responseEntity = restTemplate.exchange(
                "/full/products", HttpMethod.GET, null,
                new ParameterizedTypeReference<List<Product>>() {});

        List<Product> responseBody = responseEntity.getBody();
        assertEquals(HttpStatus.OK, responseEntity.getStatusCode());
        assertEquals(4, responseBody.size());   // Let's say we expect 4 products.
    }}
```

Summary

- The Spring Boot test ecosystem
- Writing and running tests on beans
- Mocking beans
- Additional Spring Boot test techniques