



# Spring Boot Testing

---

Leveraging Spring Boot enhancements  
for simplified integration and unit testing

## Objectives

---

After completing this lesson, you should be able to

- Enable Spring Boot Testing
- Perform integration testing
- Perform slice unit testing
- Utilize mocks to simplify unit testing

# Agenda

- **Spring Boot testing**
- Integration testing
- Web slice unit testing
- Service slice unit testing
- Repository slice unit testing



# What is Spring Boot Testing Framework?

- Provides a set of annotations and utilities for testing
  - *Examples:* `@SpringBootTest`, `@WebMvcTest`, `@MockBean`, `@DataJpaTest`
- Supports both integration and slice unit testing

# What is “Slice” Unit Testing?

- Performs isolated testing within a slice of an application
  - Web slice
  - Service slice
  - Repository slice
  - Caching slice
- Dependencies need to be mocked

# How to get Started? Add Spring Boot Test Starter

- Add the starter

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

# Testing Dependencies with spring-boot-starter-test

- **JUnit:** JUnit 4
  - In order to use JUnit 5, you have to include JUnit 5 dependencies
- **Spring Test & Spring Boot Test:** Testing annotations
- **AssertJ:** A fluent assertion library
- **Hamcrest:** A library of matcher
- **Mockito:** A Java mocking framework
- **JSONassert:** An assertion library for JSON
- **JsonPath:** XPath for JSON



# Agenda

- Spring Boot testing
- **Integration testing**
- Web slice unit testing
- Service slice unit testing
- Repository slice unit testing





# Integration Testing with `@SpringBootTest`

- Automatically searches for a `@SpringBootTestConfiguration`
  - No need to use `@ContextConfiguration`
- Provides support for different `webEnvironment` modes
  - `RANDOM_PORT`
  - `DEFINED_PORT`
  - `MOCK`
  - `NONE`
- The server gets started by the testing framework
  - Integration testing as part of CI/CD pipeline
- Auto-configures a `TestRestTemplate`

# Integration Testing with TestRestTemplate

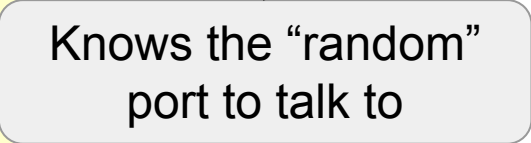
- Convenient alternative of **RestTemplate** suitable for integration tests
  - Fault tolerant: it does not throw an exception when an error response is received
  - By default configured to ignore cookies and redirects
  - Takes a relative path
- If you need customizations
  - Use **RestTemplateBuilder**
  - *Example:* add additional message converters

## Code Example with *TestRestTemplate*

```
@ExtendWith(SpringExtension.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class AccountClientBootTests {

    @Autowired
    private TestRestTemplate restTemplate;

    // Test code
}
```



Knows the “random” port to talk to

# Agenda

- Spring Boot testing
- Integration testing
- **Web slice unit testing**
- Service slice unit testing
- Repository slice unit testing



## Web Slice Testing with `@WebMvcTest`

- Disables full auto-configuration and instead apply only configuration relevant to MVC tests
- Auto-configure MockMvc testing framework
  - And optionally Spring Security
- Typically `@WebMvcTest` is used in combination with `@MockBean` for mocking its dependencies

# @Mock vs. @MockBean for Dependency

- **@Mock**
  - Use it when Spring context is not needed
- **@MockBean**
  - Use it when Spring context is needed
  - Creates a new mock bean when it is not present in the Spring context or replaces a bean with a mock bean when it is present

# Code Example with @WebMvcTest

```
@ExtendWith(SpringExtension.class)
@WebMvcTest(AccountController.class)
public class AccountControllerBootTests {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private AccountManager accountManager;

    @Test
    public void testHandleDetailsRequest() throws Exception {

        // Test code
    }
}
```



# Code Example with @WebMvcTest

Spring MVC mock testing framework

```
@Test
public void testHandleDetailsRequest() throws Exception {

    // arrange
    given(accountManager.getAccount(0L))
        .willReturn(new Account("1234567890", "John Doe"));

    // act and assert
    mockMvc.perform(get("/accounts/0"))
        .andExpect(status().isOk())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8))
        .andExpect(jsonPath("name").value("John Doe"))
        .andExpect(jsonPath("number").value("1234567890"));

    // verify
    verify(accountManager).getAccount(0L);
}
```

# Agenda

- Spring Boot testing
- Integration testing
- Web slice unit testing
- **Service slice unit testing**
- Repository slice unit testing



# Service Slice Testing

- Business logic is in the form of POJO
  - No Spring framework context used
  - Run it with **MockitoJUnitRunner**
  - Use **@Mock** for mocking dependencies
    - No need to use **@MockBean**

# Agenda

- Spring Boot testing
- Integration testing
- Web slice unit testing
- Service slice unit testing
- **Repository slice unit testing**



# Repository Slice Testing with *@DataJpaTest*

- Can be used when a test focuses only on JPA components
- Auto-configures **TestEntityManager**
  - Alternative to **EntityManager** for use in JPA tests
  - Provides a subset of **EntityManager** methods
    - Just those useful for tests
    - Helper methods for common testing tasks such as persist, flush and find
- Uses an embedded in-memory database
  - Replaces any explicit or auto-configured DataSource
  - The **@AutoConfigureTestDatabase** annotation can be used to override these settings



A man with a beard and a woman are sitting at a desk, looking at a computer monitor. The man is pointing at the screen while the woman looks on. The image is overlaid with a dark blue semi-transparent layer.

---

# ***Lab: Perform Integration and Unit Testing using Spring Boot Test***

---

**Lab project:  
44-boot-test**

**Anticipated Lab time:  
45 Minutes**