Pivotal

Advanced Testing with Spring Boot and MockMVC 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 MockMVC testing
- Perform slice testing

Agenda

- **Spring Boot testing**
- Integration testing
- Spring MVC Test Framework
- Web slice testing
- Repository slice testing
- Optional

What is Spring Boot Testing Framework?

- Built on the top of Spring Testing Framework
- Provides a set of annotations and utilities for testing
  - `@SpringBootTest`
  - `@WebMvcTest, @WebFluxTest`
  - `@DataJpaTest, @DataJdbcTest, @JdbcTest, @DataMongoTest, @DataRedisTest`
  - `@MockBean`

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 5 is default version (from Spring Boot 2.2)
- **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**
- Spring MVC Test Framework
- Web slice testing
- Repository slice testing
- Optional

Integration Testing with @SpringBootTest

- Automatically searches for **`@SpringBootConfiguration`**
  - An alternative to **`@ContextConfiguration`** for creating application context for testing
  - Use **`@SpringBootTest`** for integration testing and use **`@ContextConfiguration`** for slice testing
- Provides support for different **`webEnvironment`** modes
  - **`RANDOM_PORT, DEFINED_PORT, MOCK, NONE`**
- The embedded server gets started by the testing framework
  - Integration testing can be done as part of CI/CD pipeline
- Auto-configures a **`TestRestTemplate`**
- Meta-annotated with **`@ExtendWith`** (from Spring Boot 2.2)

Integration Testing with TestRestTemplate

- Convenient alternative of **`RestTemplate`** suitable for integration tests
  - Takes a relative path (instead of absolute path)
  - Fault tolerant: it does not throw an exception when an error response such as 404 is received from server application
  - Configured to ignore cookies and redirects

- If you need customizations
  - Use **`RestTemplateBuilder`**
  - *Example:* add custom message converters

Code Example with *TestRestTemplate*

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

   @Autowired
   private TestRestTemplate restTemplate;


    // Test code


}
```

Knows the “random” port to talk to

Code Example with *TestRestTemplate* : Test code

```
@Test
public void addAndDeleteBeneficiary() {
        String addUrl = "/accounts/{accountId}/beneficiaries";
        URI newBeneficiaryLocation = restTemplate.postForLocation(addUrl, "David", 1);
        Beneficiary newBeneficiary
              = restTemplate.getForObject(newBeneficiaryLocation, Beneficiary.class);
        assertThat(newBeneficiary.getName()).isEqualTo("David");

        restTemplate.delete(newBeneficiaryLocation);

        ResponseEntity<Account> response
             = restTemplate.getForEntity(newBeneficiaryLocation, Account.class);

        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
}
```

Relative path

Response status check

Agenda

- Spring Boot testing
- Integration testing
- **Spring MVC Test Framework**
- Web slice testing
- Repository slice testing
- Optional

Need for Spring MVC Testing

- Consider the controller below, how can you verify:
  - **`@PutMapping`** results in a correct URL mapping?
  - **`@PathVariable`** mapping is working?
  - Account is correctly mapped from incoming JSON / XML?
  - Returned status is HTTP 204?
  - Any exception is handled as expected?

```
@PutMapping("/account/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)  // 204
public void updateOrder(@RequestBody Account account, @PathVariable long id) {
  // process updated account and return empty response
  accountManager.update(id, account);
}
```

MVC Test Framework Overview

- Part of Spring Framework
  - Found in **`spring-test.jar`**

- ***Goal:*** Provide first-class support for testing Spring MVC code
  - Process requests through DispatcherServlet
  - Does *not* require running Web container to test
    - No need to coordinate server URL / port with test code

See: Spring Framework Reference, Spring MVC Test Framework
http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#spring-mvc-test-framework

Example: MockMvc Test

```
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@SpringBootTest(webEnvironment = WebEnvironment.MOCK)
@AutoConfigureMockMvc
public final class AccountControllerTests {

    @Autowired
    MockMvc mockMvc;

   @Test
   public void testBasicGet() {
      mockMvc.perform(get("/accounts"))
                  .andExpect(status().isOk());
   }
}
```

Static imports make it easier to invoke Builder & Matcher static methods

Define a MockMVC environment (Usage of @WebMvcTest would be simpler, however)

Perform tests on mockMvc instance

Setting Up Static Imports

- Static imports are key to fluid builders
  - **`MockMvcRequestBuilders.*`** and **`MockMvcResultMatchers.*`**
- You can add to Eclipse/STS 'favorite static members' in preferences
  - Java → Editor → Content Assist → Favorites
  - Add to favorite static members
    - **`org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get`**
    - **`org.springframework.test.web.servlet.result.MockMvcResultMatchers.status`**

Testing RESTful Controllers

- Argument to **`perform()`** dictates the action
  - **`get()`** (or **`put()`**, **`post()`**, etc.) from **MockMvcRequestBuilders**
  - Append with methods from **MockHttpServletRequestBuilders**

```
@Test
public void testRestfulGet() throws Exception {
  mockMvc.perform(
                 get("/accounts/{acctId}","123456001")
                .accept(MediaType.APPLICATION_JSON))
               …
               //  Continued …
}
```

`MockMvcRequestBuilders` methods go inside **`perform()`**

MockMvcRequestBuilders

Static Methods

- Standard HTTP get, put, post, delete operations
  - *fileUpload* also supported
  - Argument usually a URI template string
  - Returns a **`MockHttpServletRequestBuilder`** instance (for chaining other methods)

```
// Perform a get using URI template style
mockMvc.perform(get("/accounts/{acctId}", "123456001"))

// Perform a get using request parameter style
mockMvc.perform(get("/accounts?myParam={acctId}", "123456001"))
```

MockHttpServletRequestBuilder

Static Methods

Method	Description
param	Add a request parameter – such as param(“myParam”, 123)
requestAttr	Add an object as a request attribute. Also, sessionAttr does the same for session scoped objects
header	Add a header variable to the request. Also see headers, which adds multiple headers
content	Request body
contentType	Set content type (Mime type) for body of the request
accept	Set the requested type (Mime type) for the expected response
locale	Set the local for making requests

Testing RESTful Controllers

- **`perform()`** returns **`ResultActions`** object
  - Can chain **`expects`** together – fluid syntax
  - **`content()`** and **`jsonPath()`** from **`MockMvcResultMatchers`**

```
@Test
public void testRestfulGet() throws Exception {
  mockMvc.perform(
                 get("/accounts/{acctId}","123456001")
                .accept(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(content().contentType("application/json"));
}
```

`MockMvcResultMatchers` methods go inside **`andExpect()`**

MockMvcResultMatchers

Static Methods

- Returns Matchers providing specific assertions
  - Uses Hamcrest, see JavaDoc for details

Method	Matcher Returned	Description
content	ContentResultMatchers	Assertions relating to the HTTP response body
header	HeaderResultMatchers	Assertions on the HTTP headers
status	StatusResultMatchers	Assertions on the HTTP status
xpath		Search returned XML using Xpath expression
jsonPath		Search returned JSON using JsonPath

MockHttpServletRequestBuilder Examples

- Setting Accept Header

```
mockMvc.perform(get("/accounts/{acctId}","123456001")
                        .accept("application/json")  // Request JSON Response
            ...
```

- PUTting JSON payload

```
mockMvc.perform(put("/accounts/{acctId}","123456001")
                        .content("{        …        }")
                        .contentType( "application/json")
            …
```

Printing Debug Information

- Sometimes you want to know what happened
  - **`andDo()`** performs action on **`MvcResult`**
  - **`print()`** sends the **`MvcResult`** to output stream
  - Or use **`andReturn()`** to get the **`MvcResult`** object

```
// Use this to access the print() method
import static org.springframework.test.web.servlet.result.MockMvcResult.*;

// Other static imports as well
// Use print() method in test to get debug information
mockMvc.perform(get("/accounts/{acctId}","123456001"))
        .andDo(print())      // Add this line to print debug info to the console
        .andExpect(status().isOK())
        …
```

Agenda

- Spring Boot testing
- Integration testing
- Spring MVC Test Framework
- **Web slice testing**
- Repository slice testing
- Optional

What is “Slice” Testing?

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

Web Slice Testing with @WebMvcTest

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

@Mock vs. @MockBean for Dependency

- `@Mock`
  - From Mockito framework
  - Use it when Spring context is not needed

- `@MockBean`
  - From Spring Boot Framework
  - 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

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

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private AccountManager accountManager;

    @Test
    public void testHandleDetailsRequest() throws Exception {

        // Test code
    }
}
```

Only creates beans relevant to AccountController

Code Example with @WebMvcTest

Programming the MockBean

```
@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)
            .andExpect(jsonPath("name").value("John Doe"))
            .andExpect(jsonPath("number").value("1234567890"));

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

Agenda

- Spring Boot testing
- Integration testing
- Spring MVC Test Framework
- Web slice testing
- **Repository slice testing**
- Optional

Repository Slice Testing with *@DataJpaTest*

- Can be used when a test focuses only on JPA components
  - Loads @Repository beans, excludes all other @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 persistFlushFind(), persistAndFlush()
- Uses an embedded in-memory database
  - Replaces any explicit or auto-configured DataSource
  - The **`@AutoConfigureTestDatabase`** annotation can be used to override these settings

Summary

- @SpringBootTest expands options for testing
- Spring MVC Test framework provides a mock web environment
  - No need for running an external app server.
- Boot provides web slice testing
  - Mock MVC test focused on specific controller
- Boot provides JPA slice testing

Lab: Perform Integration and Unit Testing using Spring Boot Test

Lab project:
40-boot-test

Anticipated Lab time:
45 Minutes

Agenda

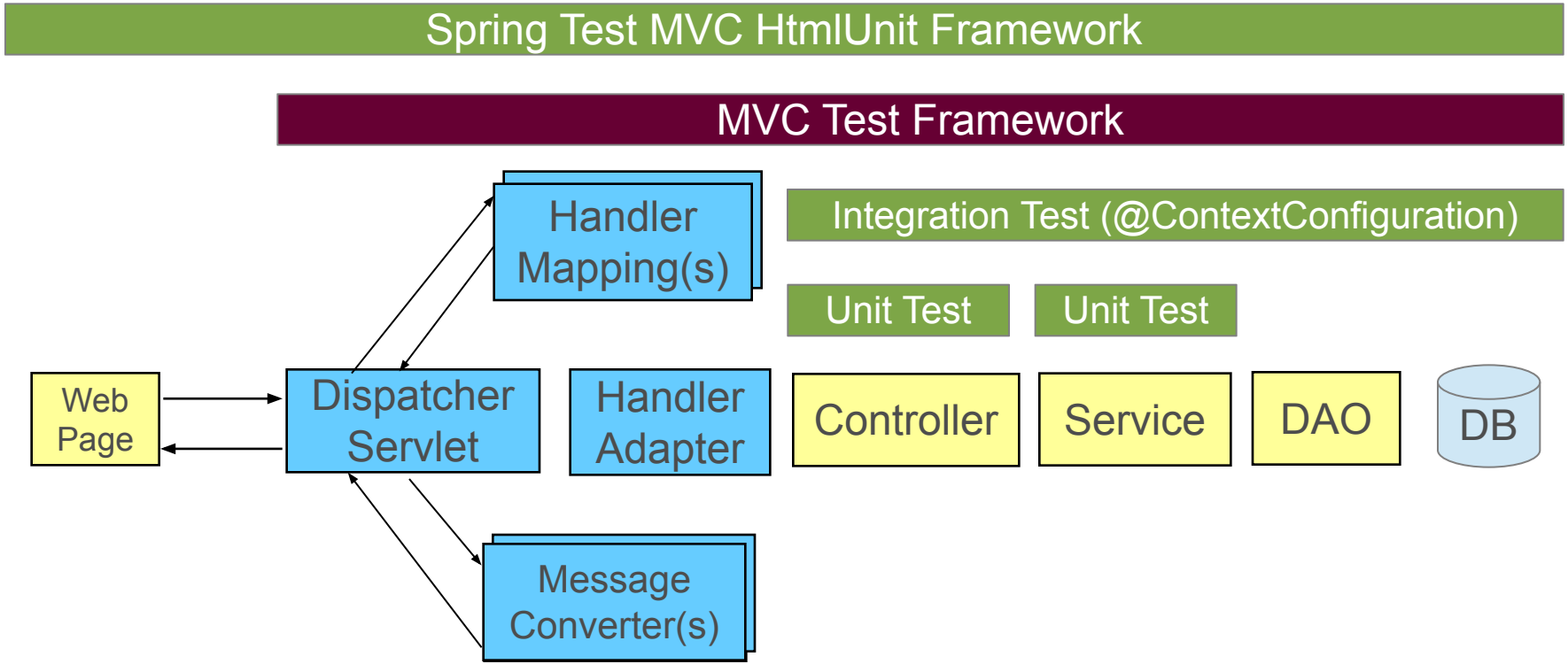
- Spring Boot testing
- Integration testing
- Spring MVC Test Framework
- Web slice testing
- Repository slice testing
- **Optional**
  - **Traditional Web Testing**

Testing Traditional Web Pages

- Server-side web pages from Spring MVC easy to test

```
@Test
public void basicAccountDetailsRequest() {
 mockMvc.perform(get("/accounts/{acctId}", "123456001"))
            .andExpect(status().isOk())                    // Expect status code 200
            .andExpect(model().size(1))                    // Expect 1 model attribute
            .andExpect(view().name("accounts/show"))       // Expect this view returned
            .andExpect(forwardedUrl("/WEB-INF/accounts/show.jsp"));
}
```

Testing Each Layer



What Is It For?

- Easily test web-pages using familiar tools
  - Integration testing without starting an application server
  - Supports HtmlUnit, WebDriver, and Geb
  - Support testing of JavaScript
  - Optionally test using mock services for faster testing
- **Note:** MockMvc only renders content with view technologies that do not rely on a Servlet Container
  - Thymeleaf, Freemarker, Velocity, …
  - Does not render JSPs or Tiles

Setup an HttpUnit WebClient

- Integrates WebClient with MockMvc
  - Requests to *localhost* processed “out-of-server” by MockMVC framework

```
@Autowired
WebApplicationContext context;

WebClient webClient;

@Before
public void setup() {
        webClient = MockMvcWebClientBuilder
                .webAppContextSetup(context).build();
}
```

Run a Test

- Use WebClient in the usual way

```
HtmlPage accountPage =
        webClient.getPage("http://localhost/accounts/123456789");
```

- Request processed by your Spring MVC application
  - Without using a container

Submit a Form

- 100% HttpUnit – no Spring

```
HtmlPage searchPage = webClient.getPage("http://localhost/accounts/search");

HtmlForm form = searchPage.getHtmlElementById("searchForm");
HtmlTextInput summaryInput = searchPage.getHtmlElementById("search");
summaryInput.setValueAttribute("Keith");

HtmlSubmitInput submit =
        form.getOneHtmlElementByAttribute("input", "type", "submit");
HtmlPage accountPage = submit.click();
```

Verify Result

- Again, standard HttpUnit

```
// Ensure the page returned is the right one
assertThat(accountPage.getUrl().
   toString()).endsWith("/accounts/123456789");

// Check account has the right name
String name = accountPage.getHtmlElementById("name").getTextContent();
assertThat(name).isEqualTo("Keith and Keri Donald");
```

http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#spring-mvc-test-server-resources

Additional Testing Capabilities

- Testing content negotiation
- Client side can be tested as well

- Additional references
  - Spring Reference Guide – Testing Chapter
    http://docs.spring.io/spring/docs/current/spring-framework-reference/html/testing.html
  - Spring Boot Testing
    https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html