



Testing with Spring's MockMVC Framework

Test the full MVC Stack in a JUnit test

Objectives

After completing this lesson, you should be able to

- Test the various Web Application components
- Describe the Spring MVC Test Framework
- Test Spring Boot applications

Agenda

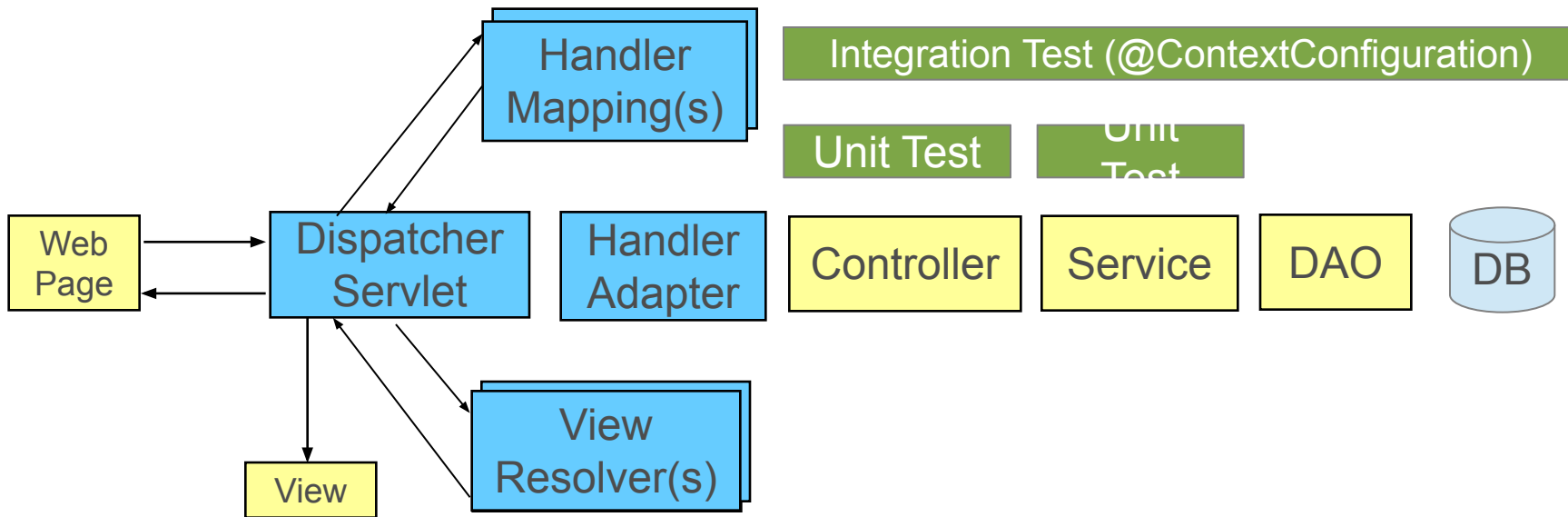
- **Testing Web Applications**
- Spring MVC Test Framework
- Going Further
- Using HtmlUnit for Spring Testing
- Lab



Testing Each Layer

Spring Test MVC HtmlUnit Framework

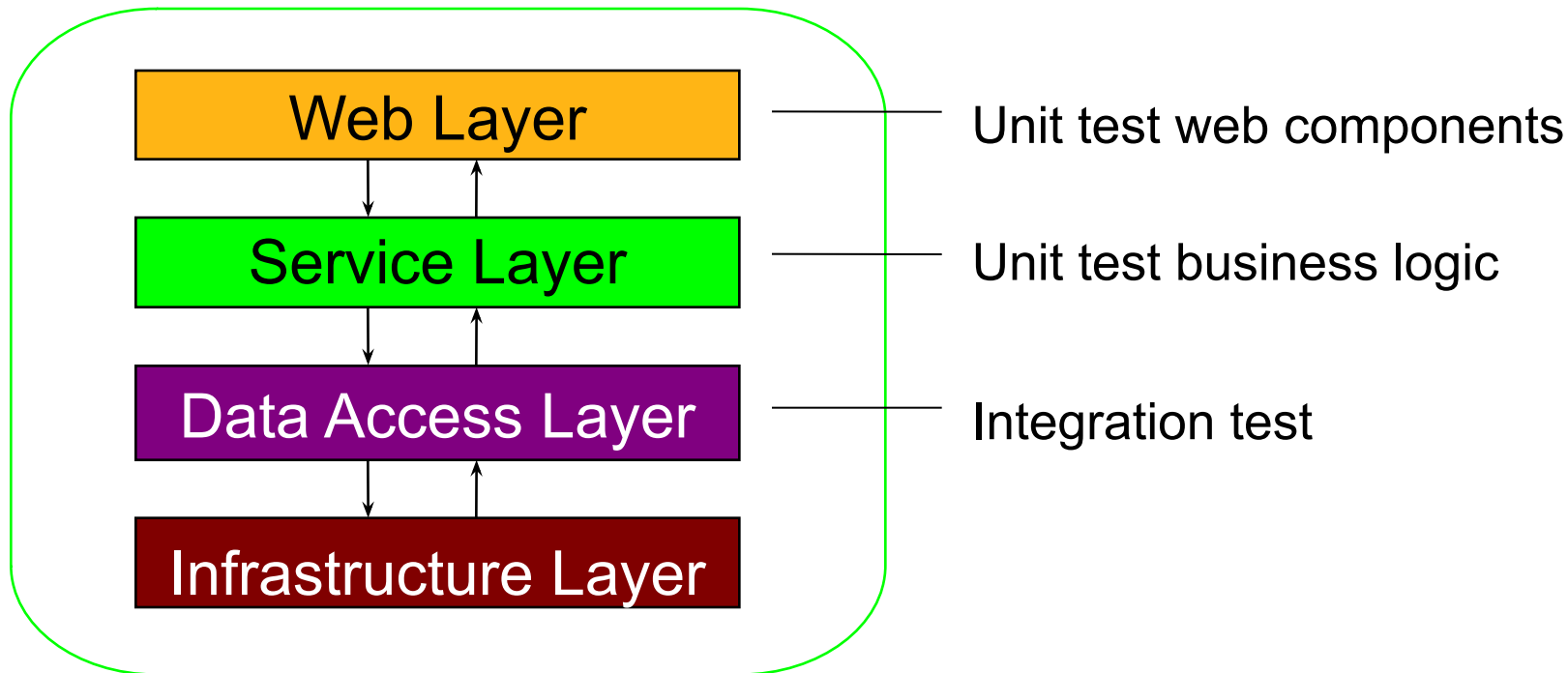
MVC Test Framework



Testing the Web Layer

- **Unit Tests**
 - One method at a time
 - Isolate dependencies (Mocking, Stubbing)
- **Integration Tests**
 - Component interaction scenarios
 - Configure dependencies with Spring
- **End to End Tests**
 - Exercise the application's user interface
 - Fully integrated system

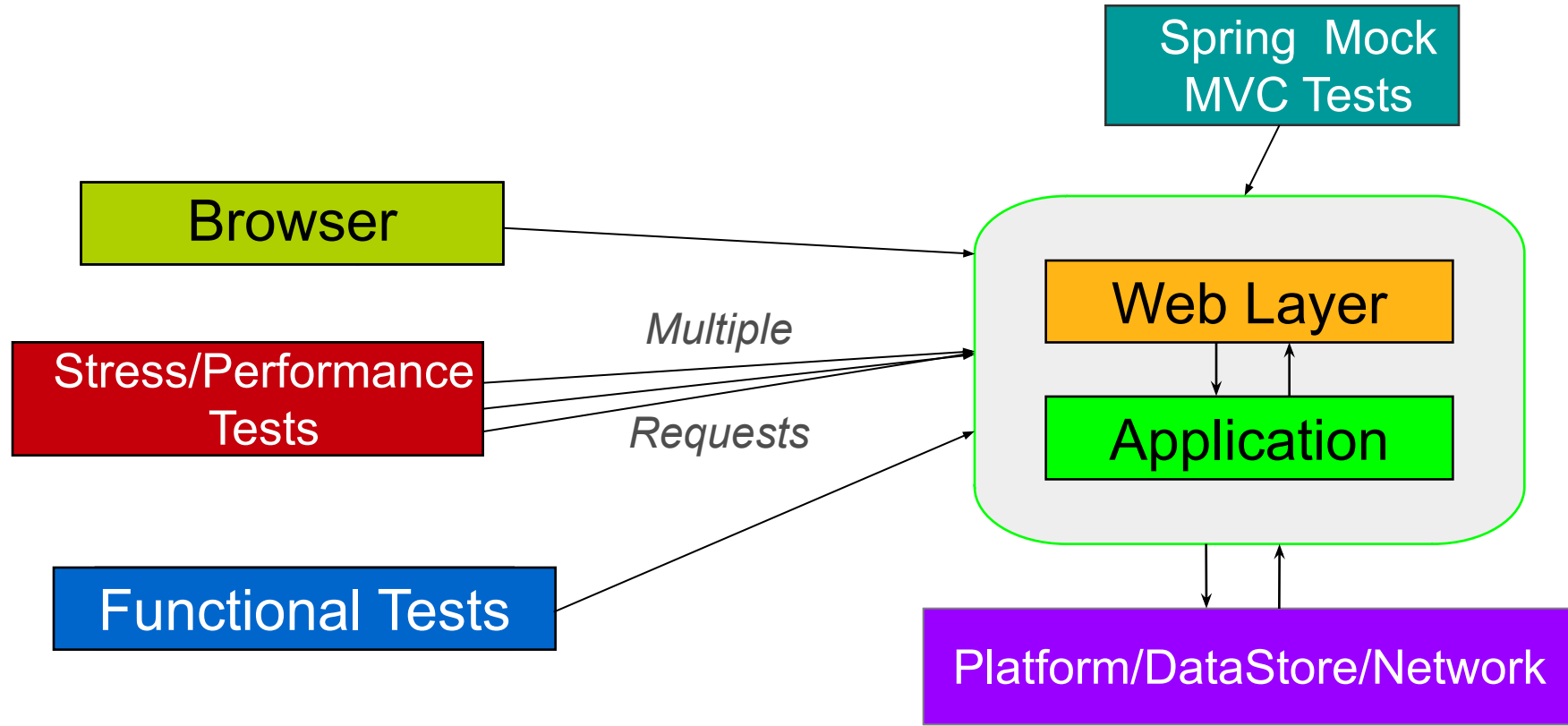
Testing Each Layer



Testing the Web Layer

- Unit test web components
 - controllers, interceptors, custom views
- Use stubs or mocks to isolate services
 - run fast and often
- What to test?
 - the logic adapting HTML requests to the service layer
 - input parameters, validation, etc.

Testing The Whole System



What Still Needs Verification?

- Client-side behavior
 - Browser/UI interactions
- End-To-End Testing
 - Exercise application running as a fully integrated system
 - Verify “no surprises” from underlying code
 - Performance/Load/Stress testing
- **Server-side configuration**
 - **Servlet container**
 - **Frameworks (Spring MVC)**

Browser

Functional Tests

Stress/Performance
Tests

← *This section*

Spring Mock MVC Tests

Agenda

- Testing Web Applications
- **Spring MVC Test Framework**
- Going Further
- Using `HtmlUnit` for Spring Testing
- Lab



Need for Spring MVC Testing

- Consider the controller below, how can you verify:
 - `@PostMapping` results in a correct URL mapping?
 - `@Valid` is active? Expected validation checks happen?
 - Framework redirects where you expect?
 - ViewResolver chain translates “`accounts/edit`” to the correct View?
 - Any exception results in a display of the error page?

```
@PostMapping(path="/account")
public String save(@Valid Account acc, BindingResult result) {
    if (result.hasErrors()) return "accounts/edit";

    accountManager.update(acc);
    return "redirect:" + acc();
}
```

MVC Test Framework Overview



SPRING TEST
FRAMEWORK

- Part of Spring Framework
 - Found in `spring-test.jar`
- **Goal:** Provide first-class support for testing Spring MVC code
 - Fluent API
 - Process requests through DispatcherServlet
 - Does *not* require Web container to test



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

<http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#spring-mvc-test-framework>

Example

```
@ExtendWith(SpringExtension.class)
@SpringBootTest(classes=MyApp.class)
public final class AccountControllerTests {
```

```
    @Autowired
```

```
    private WebApplicationContext context;
```

```
    private MockMvc mockMvc;
```

```
    @BeforeEach
```

```
    public void setup() {
        mockMvc = MockMvcBuilders.webAppContextSetup(context).build();
    }
```

```
    @Test
```

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

Setup the test – defaults to a Web-ApplicationContext

Inject application context

Define MockMvc field and initialize it

Perform tests on mockMvc instance

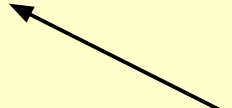
Unit Testing vs MVC Test Framework

- Unit Testing
 - Use to test logic of controllers
 - Create & populate model, test for proper views, model data, etc
- MVC Test Framework
 - Use to test MVC annotations (like `@RequestMapping`, `@PathVariable`, etc)
 - Also can perform integration tests with other Spring MVC beans such as View Resolvers, Interceptors, Filters ...

Setting Up The Test Harness

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

```
public final class AccountControllerTests {  
    @Autowired  
    private WebApplicationContext wac;  
    private MockMvc mockMvc;  
  
    @Before  
    public void setup() {  
        // Initialize mockMvc using WebApplicationContext  
        mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();  
    }  
  
    // Now perform tests using the mockMvc object  
}
```



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

Setting Up Static Imports

- Static imports are key to fluid builders
 - `MockMvcRequestBuilders.*` and `MockMvcResultMatchers.*`
- You can add to Eclipse '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`

Perform and Expect

MockMvcRequestBuilders
methods go in `perform()`

- Argument to `perform()` dictates the action
 - `perform()` returns **ResultActions** object
 - Can chain **expects** together – fluid syntax

Options: *get, put, post, delete, fileUpload*

```
@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"));
}
```

MockMvcResultMatchers
methods in *italics*

Matcher specific *assertion methods* go here

MockMvcRequestBuilders

Static Methods

- Standard HTTP get, put, post, delete operations
 - *fileUpload* also supported
 - Argument usually a URI template string
 - Returns a **MockHttpServletRequestBuilder** instance

```
// 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
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

MockHttpServletRequestBuilder Examples

- Adding Request Parameters

```
mockMvc.perform(post("/accounts/{acctId}", "123456001")
    .param("name", "Some Name"           // Simulate form submission
    .param("dateOfBirth", "2012-05-10")
    ... // More parameters and/or other MockHttpServletRequestBuilder
        // methods can be added
```

- Setting Accept Header

```
mockMvc.perform(get("/accounts/{acctId}", "123456001")
    .accept("application/json;charset=UTF-8") // Request JSON Response
    ...
```

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
model	ModelResultMatchers	Assertions relating to the returned Model
header	HeaderResultMatchers	Assertions on HTTP headers
view	ViewResultMatchers	Assertions on returned view
status	StatusResultMatchers	Assertions on the HTTP status
forwardedUrl	ResultMatcher	Assertions on forwarded URL



Warning: “content” does *not* contain JSP output when running outside a container!

Simulating Complete Form Submission

```
@Test public void testPutAccountSuccess() {  
    mockMvc.perform( put("/accounts/{acctId}", "123456001")  
        .param("name", "Some Name") // Simulate form submission  
        .param("dateOfBirth", "2012-05-10")  
        .param("email", "testEmail@someaddress.com")  
        .param("receiveNewsletter", "1")  
        .param("receiveMonthlyEmailUpdate", "1") )  
        .andExpect(status().isFound()) // Because this is a redirect  
        .andExpect(redirectedUrl("123456001"));  
}
```

Redirect URL
from controller

```
@PutMapping("/accounts/{acctId}")  
public String save(@Valid Account account, BindingResult result) {  
    if (result.hasErrors()) { return "accounts/edit"; }  
    accountManager.update(account);  
    return "redirect:" + account.getNumber();  
}
```

Checking for Validation and Binding Errors

@Test

Testing same Controller method

```
public void testPutAccountFailValidation() throws Exception {  
    mockMvc.perform( put("/accounts/{acctId}", "123456001")  
        .param("email", "bogusemail")  
        ... // Other parameters as previous slide  
        .andExpect(model().attributeHasErrors("account"))  
        .andExpect(model().attributeErrorCount("account", 1))  
        .andExpect(model().attributeHasFieldErrors("account", "email"))  
        .andExpect(status().isOk())  
        // Because this is a forward to the same view  
        .andExpect(view().name("accounts/edit"));  
        // The redirect URL provided by the controller  
    }  
}
```

Bad email – error expected

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
    ...
```


Agenda

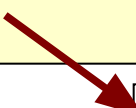
- Testing Web Applications
- Spring MVC Test Framework
- **Going Further**
- Using HtmlUnit for Spring Testing
- Lab
- Optional/Advanced ...



Testing RESTful Controllers I

- Can also test RESTful interactions
 - Need to specify expected contentType and accept values for representation

```
@GetMapping(produces="application/json")  
public @RequestBody Account get(@PathVariable String number) {  
    return accountManager.findAccount(number);  
}
```



```
{  
    "entityId": 1,  
    "number": "123456001",  
    "creditCardNumber": "1234123412340001",  
    ...  
}
```

JSON representation

Testing RESTful Controllers II

- Note use of `accept()` and `jsonPath()`
 - Assertions use JsonPath – like XPath for JSON
 - See <https://code.google.com/p/json-path>

@Test

```
public void testRestfulGet() throws Exception {  
    mockMvc.perform(get("/accounts/{acctId}", "123456001")  
        .accept(MediaType.APPLICATION_JSON))  
        .andExpect(content().contentType("application/json"))  
        .andExpect(jsonPath("$.number").value("123456001"));  
}
```



For more on JsonPath see: <https://code.google.com/p/json-path>

Spring Boot Testing



- Spring Boot provides additional testing options
 - **@WebMvcTest**
 - Component scanner only creates *web* beans
 - `@Controller`, `@ControllerAdvice`, `Filter`, `@JsonComponent`, `WebMvcConfigurer` ...
 - No Service, Repository beans created
 - **@MockBean**
 - Define mocks
 - *Example:* mock the service(s) your Controller(s) rely on

Testing Controllers with Mocks

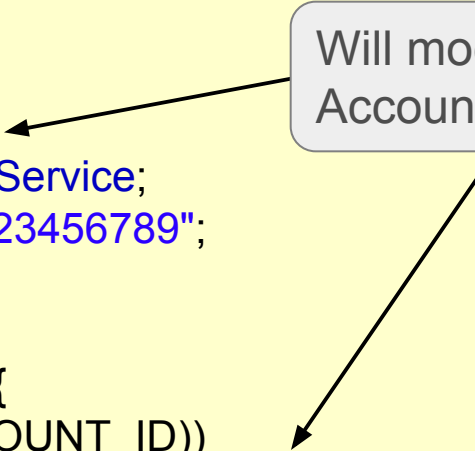


```
@ExtendWith(SpringExtension.class)
@SpringBootTest(AccountController.class)
public class MyControllerTests {
    @Autowired private MockMvc mvc;
    @MockBean private AccountService accountService;
    private static final String ACCOUNT_ID = "123456789";

    @Test
    public void testExample() throws Exception {
        given(this.accountService.getAccount(ACCOUNT_ID))
            .willReturn(new Account(ACCOUNT_ID, "Keith and Keri Donald"));

        mvc.perform(get("/accounts/{id}", ACCOUNT_ID))
            .accept(MediaType.TEXT_PLAIN)
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.number").value(ACCOUNT_ID));
    }
}
```

Will mock the Account Service



Performing Security Testing – 1

```
public class AccountControllerIntegrationTests {  
    @Autowired private WebApplicationContext context;  
    @Autowired private FilterChainProxy springSecurityFilterChain;  
    private MockMvc mockMvc;  
  
    @Before  
    public void setup() {  
        mockMvc = webAppContextSetup(context).  
            addFilter(springSecurityFilterChain).build();  
    }  
    @Test  
    public void requiresAuthentication() throws Exception {  
        mockMvc.perform(get("/"))  
            .andExpect(redirectedUrl("http://localhost/login"));  
    }  
}
```

Inject **SpringSecurityFilterChain** and add as a filter

Assert unauthenticated access redirects to the configured login page

Performing Security Testing – 2

```
import static
```

```
org.springframework.security.test.web.servlet.setup.SecurityMockMvcConfigurers.*;
```

```
public class AccountControllerIntegrationTests {
```

```
@Autowired private WebApplicationContext context;
```

```
private MockMvc mockMvc;
```

```
@BeforeEach
```

```
public void setup() {
```

```
    mockMvc = webAppContextSetup(context).apply(springSecurity()).build();
```

```
}
```

```
@Test
```

```
@WithMockUser(username="admin",roles={"USER","ADMIN"})
```

```
public void requiresAuthentication() throws Exception {
```

```
    mockMvc.perform(get("/")).andExpect(status().isOk());
```


```
}
```

```
}
```

Adds **springSecurity**
FilterChain as a filter



Run with this user
having roles



No redirect this time – logged in as “admin” with required role



Agenda

- Testing Web Applications
- Spring MVC Test Framework
- Going Further
- **Using HtmlUnit for Spring Testing**
- Lab



What Is It For?

HtmlUnit



- 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 works fully 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/html/single/#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>

Summary

- After completing this lesson, you should have learned that:
- Browsers provide many tools for developers
 - JavaScript debugging, logging
- Web application testing is essential part of a developer's responsibilities
- The Spring MVC Test framework offers an enhanced ability to test not just method calls but annotations, view resolution, status codes and more