Introduction to Spring MVC Test Framework

Avoid the danger which has not yet come

Test Driven Development

- The strictest definition of Test Driven Development (TDD) is to always write the tests first, then the code.
- A looser interpretation is Test Oriented Development (TOD), where we alternate between writing code and tests as part of the development process.
- The most important thing is for a codebase to have as complete a set of unit tests as possible
- The quality of tests is always higher when they are written at around the same time as the code that is being developed

Testing is Essential

- Industry Average: "about 15 - 50 errors per 1000 lines of ***delivered code***." [1980]

- Microsoft Applications: finds "about 10 - 20 defects per 1000 lines of code ***during in-house testing***. [1992]

SDI – "Star Wars"

- …the X-rays emitted by an exploding H-bomb would be focused on the travelling missile. One of the many technical issues was that the device involved some millions of lines of computer code, and could not be tested in advance. ***One computer expert said that the maximum number of lines of untested code without a bug was about 9;***

Unit Tests / Unit Testing

- **Code written to test code under test**
  - Designed to test specific sections of code
  - Percentage of lines of code tested is code coverage
    - Ideal coverage is in the 70-80% range
  - Should be 'unity' and execute very fast
  - Should have no external dependencies
    - ie no database, no Spring context, etc

Integration Tests

- Designed to test behaviors between objects and parts of the overall system
  - Much larger scope
  - Can include the Spring Context, database, and message brokers
  - Will run much slower than unit tests

Functional Tests

- Typically means you are testing the running application
  - Application is live, likely deployed in a known environment
  - Functional touch points are tested
    - i.e. Using a web driver, calling web services, sending / receiving messages, etc

Test Scope Dependencies

- Using spring-boot-starter-test (default from Spring Initializr will load the following dependencies:
  - JUnit - The de-facto standard for unit testing Java applications
  - Spring Test and Spring Boot Test - Utilities and integration test support for Spring Boot applications
  - AssertJ - A fluent assertion library
  - Hamcrest - A library of matcher objects
  - Mockito - A Java mocking framework
  - JSONassert - An assertion library for JSON
  - JSONPath - XPath for JSON

Test Doubles

- *any kind of pretend object used in place of a real object for testing purposes.*

TYPES

- **Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists.
- **Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example).
- **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
- **Mocks** objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

ProductService Interface

```
public interface ProductService   {
   // Returns a List of all Products
  public List<Product> getAll();
  // Save a Product
  public void save(Product product);
  // Find a Product by id
  public Product findOne(Long id);

}
```

Here is our Interface We want to MOCK it!

ProductService - Behavior

```
public class ProductServiceImpl implements ProductService {

   @Autowired
   ProductRepository productRepository;

   public List<Product> getAll() {
      return productRepository.getAll();
   }

   public void save(Product product) {
       productRepository.save(product);
       return ;
   }

   public Product findOne(Long id) {
      return productRepository.findOne( id);
   }
}
```

This is the behavior to Mock!
See Product4aTestSA Demo

Mockito

Tasty mocking framework for unit tests in Java

Mockito

Enables mock creation, verification and stubbing.

Creates simulated objects that mimic the behavior of real objects in controlled ways.

Allows specification of which, and in what order, methods will be invoked on a mock object and what parameters will be passed to them, as well as what values will be returned.

```
// Need data to mock Product Repository getAll
ListBuilder listBuilder = new ListBuilder();
// Declare mock Product Repository getAll
when(productRepositoryMock.getAll()).thenReturn(listBuilder.getProductList());
 // Invoke getAll
List<Product> products = productService.getAll();
// Validate results ….
```

with HAMCREST…

Hamcrest

"Matchers that can be combined to create flexible expressions of intent"

Framework for 'match' rules to be defined declaratively.

Strives to make your tests as readable as possible.

Integrates with Junit ;TestNG…and Mock frameworks

"Third Generation" Matcher framework

JUnit's ***assertEquals*** replaced by Hamcrest ***assertThat***

```
assertThat(products, hasItem(
          allOf(
                hasProperty("id", is(1L)),
                hasProperty("category", hasProperty("name", is("Sports"))),
                hasProperty("description", is("Two wheels")),
                hasProperty("name", is("Bicycle"))
               )
        ));
```

Hamcrest

Organizing Unit Test Data
Object Mother

Object Mother is a set of factory methods that allow us creating similar objects in tests

Customer customer = CreateTestCustomer();

Becomes

Customer customer = CustomerObjectMother.CreateCustomer();

Issue

Names could become Long & Ambiguous depending on the number of variables

Customer customer =
CustomerObjectMother.CreateWashingtonBasedCustomer();

Versus

Customer customer = CustomerObjectMother.CreateCustomer();
customer.State = new State("WA");

More data clarity at test level

Organizing Unit Test Data
Test Data Builder

Based on Builder Pattern

[Solution to the telescoping constructor anti-pattern]

Reduces the number of constructors, by processing initialization parameters step by step

```
Customer customer = new CustomerBuilder()
                        .withState("WA")
                        .withZipCode("98765")
                        .withFirstName("Fred")
                        .build();
```

CategoryBuilder

```
public class CategoryBuilder {
    private Category category;

    public CategoryBuilder() {
        this.category = new Category();
    }

    public CategoryBuilder withName(String name) {
        this.category.setName(name);
        return this;
    }

    public CategoryBuilder withId(Integer id) {
        this.category.setId(id);
        return this;
    }

    public Category build() {
        return category;
    }
}
```

Spring Framework Test

- The adoption of the test-driven-development (TDD) approach to software development is advocated by the Spring team:

  By application of the IoC principle to unit testing

  & support for integration testing

- Built-in Mock Libraries:

  org.springframework.mock.env
  org.springframework.mock.jndi
  org.springframework.mock.web
  org.springframework.mock.web.portlet

Spring Test Reference

Spring MVC Test

- First class support for testing Spring MVC code
- Built on top of the Servlet API mock objects:
- **org.springframework.mock.web**
  - comprehensive set of Servlet API mock objects, which are useful for testing web contexts, controllers, and filters.

THEREFORE: does *not* use a running Servlet container.

Mocks the DispatcherServlet for full Spring MVC runtime behavior Support for :

Standalone mode - instantiate and test individual controllers

Application Context mode - test in full configuration environment