Testing by Example with Spring 2.5

Sam Brannen
Senior Software Engineer
sam.brannen@springsource.com
http://www.springsource.com

Goals of the Presentation

- Gain an overview of the new Spring TestContext Framework
- Learn how to get the most out of Spring's testing support

Before we start…

… a show of hands …

- Unit and integration testing
- Integration testing with Spring
- Spring 2.5
- Spring TestContext Framework
- JUnit 3.8, JUnit 4.x, TestNG

Agenda

- Background Information
- The TestContext Framework
- Examples and Custom Solutions

Background Information

A bit of history and the impetus for change

History of Testing in Java

- **JUnit**
  - Developed by Kent Beck (XP) and Erich Gamma (GoF)
  - Debuted on the Java scene → 2000
  - Wide adoption through JUnit 3.8.x
- **TestNG**
  - *Modern* unit and integration testing framework developed by Cédric Beust and Alexandru Popescu
  - Debuted in April 2004
- **Java SE 5** → September 2004
- **JUnit 4** → 2006 (JUnit 4.4 → July '07)

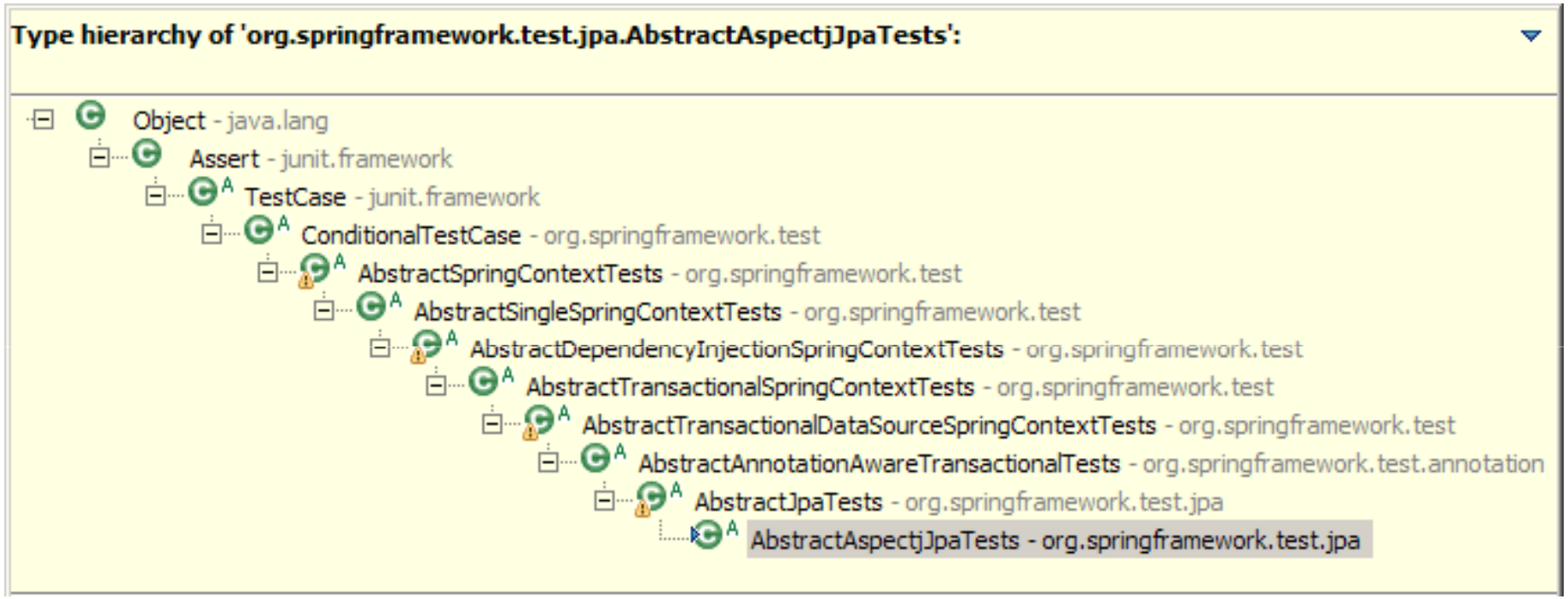
Spring and Unit Testing

- POJO-based programming model
  - Program to interfaces
  - IoC / Dependency Injection
  - Out-of-container testability
- Testing mocks/stubs for various APIs:
  - Servlet
  - Portlet
  - JNDI

Spring and Integration Testing

- Context management & caching
- Dependency injection of test instances
- Transactional test management
- Abstract JUnit 3.8 based support classes
  - Abstract*SpringContextTests

Growing Pains of an Inheritance Hierarchy



2 from JUnit + **9** from Spring = **11 classes!**

Impetus for Change

Out with the old and in with the new!

- Break free of the inheritance hierarchy
- Provide an easy migration path
- Maintain the existing feature set
- Allow for extensibility
- Provide consistent support across testing frameworks

The TestContext Framework

From AbstractDependencyInjection-SpringContextTests to @ContextConfiguration

Spring TestContext Framework: In a Nutshell

- Fully revised, annotation-based test framework
- Supports JUnit 4.4 and TestNG as well as JUnit 3.8
- Supersedes older JUnit 3.8 base classes
  - AbstractDependencyInjection-SpringContextTests & friends
  - They're still there for JDK 1.4

Spring TestContext Framework: Goals (1)

- Convention over configuration
  - Use only annotations
  - Reasonable defaults that can be overridden
- Consistent support for Spring's core annotations
- Spring-specific integration testing functionality:
  - Context management & caching
  - Dependency injection of tests
  - Transactional test management

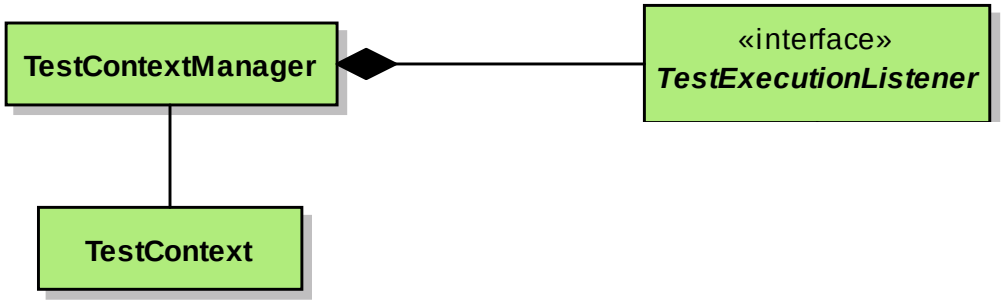
Spring TestContext Framework: Goals (2)

- Agnostic of the testing framework in use
- Support unit and integration testing
- Extensible and highly configurable
- Support POJO test classes
- Provide base support classes where necessary and/or as a convenience

TestContext Key Abstractions (1)

- **`TestContext`**
  - Context for executing test
- **`TestContextManager`**
  - Signals events for the managed context to listeners
- **`TestExecutionListener`**
  - Reacts to test execution events
  - Spring provides these:
    - DependencyInjectionTestExecutionListener
    - DirtiesContextTestExecutionListener
    - TransactionalTestExecutionListener

TestContext Key Abstractions (2)

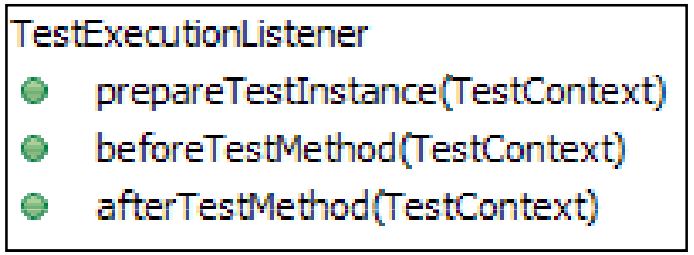


TestExecutionListener Ordering and Nesting


```
@TestExecutionListeners({
  DependencyInjectionTestExecutionListener.class,
  DirtiesContextTestExecutionListener.class,
  TransactionalTestExecutionListener.class})
```

```
{1} prepareTestInstance()
{2} prepareTestInstance()
{3} prepareTestInstance()

{1} beforeTestMethod()
{2} beforeTestMethod()
{3} beforeTestMethod()
{*} test method
{3} afterTestMethod()
{2} afterTestMethod()
{1} afterTestMethod()
```



Annotated Test Class Example


```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
// defaults to MyTests-context.xml in same package
public class MyTests {

  @Autowired
  private MyService myService;

  @BeforeTransaction
  public void verifyInitialDataba

  @Before
  public void setUpTestDataWithin

  @Test
  public void myTest() { … }

  @Test
  @Transactional
  public void myOtherTest() { … }
}
```

"The Spring TestContext Framework is an excellent example of good annotation usage as it allows composition rather than inheritance."

- Costin Leau, SpringSource

TestContext Annotations

- TestExecutionListeners
  - **`@TestExecutionListeners`**
- Application Contexts
  - **`@ContextConfiguration`** and **`@DirtiesContext`**
- Dependency Injection
  - **`@Autowired`**, **`@Qualifier`**, **`@Resource`**, **`@Required`**, etc.
- Transactions
  - **`@Transactional`**, **`@NotTransactional`**, **`@TransactionConfiguration`**, **`@Rollback`**, **`@BeforeTransaction`**, and **`@AfterTransaction`**

TestContext Annotations - JUnit

- Testing Profiles
  - **`@IfProfileValue`** and **`@ProfileValueSourceConfiguration`**
- JUnit extensions
  - **`@ExpectedException`**, **`@Timed`**, **`@Repeat`**

Using the TestContext Framework

- Use the **`SpringJUnit4ClassRunner`** for JUnit 4.4 or instrument test class with **`TestContextManager`** for TestNG
- Or extend one of the new base classes
  - **`Abstract(Transactional)[JUnit38|Junit4|TestNG]SpringContextTests`**

Test Helper classes

- SimpleJdbcTestUtils
  - Count table rows, empty tables, run SQL script
- ReflectionTestUtils
  - Set non-public fields, invoke non-public setters
- Independent of TestContext framework

Examples and Custom Solutions

PetClinic: JUnit 3.8 - Legacy


```
public class JUnit38LegacyClinicTests extends
    AbstractTransactionalDataSourceSpringContextTests {

  private Clinic clinic;

  public void setClinic(Clinic clinic) {
    this.clinic = clinic;
  }

  protected String getConfigPath() {
    return "/clinic-context.xml";
  }

  public void testGetVets() {
    Collection<Vet> vets = this.clinic.getVets();
    assertEquals("JDBC query must match",
      super.countRowsInTable("VETS"), vets.size());
  }
}
```

PetClinic: JUnit 3.8 - TestContext

```
@ContextConfiguration(locations="/clinic-context.xml")
public class JUnit38ClinicTests extends
    AbstractTransactionalJUnit38SpringContextTests {

  @Autowired
  protected Clinic clinic;

  // Test methods must be named test*()
  public void testGetVets() {
    Collection<Vet> vets = this.clinic.getVets();
    assertEquals("JDBC query must match",
      super.countRowsInTable("VETS"), vets.size());
  }
}
```

PetClinic: JUnit 4.4 - TestContext


```
@ContextConfiguration(locations="/clinic-context.xml")
public class JUnit4ClinicTests extends
    AbstractTransactionalJUnit4SpringContextTests {

  @Autowired
  protected Clinic clinic;

  @org.junit.Test
  public void getVets() {
    Collection<Vet> vets = this.clinic.getVets();
    assertEquals("JDBC query must match",
      super.countRowsInTable("VETS"), vets.size());
  }
}
```

PetClinic: TestNG - TestContext

```
@ContextConfiguration(locations="/clinic-context.xml")
public class TestNGClinicTests extends
    AbstractTransactionalTestNGSpringContextTests {

  @Autowired
  protected Clinic clinic;

  @org.testng.annotations.Test
  public void getVets() {
    Collection<Vet> vets = this.clinic.getVets();
    assertEquals("JDBC query must match",
      super.countRowsInTable("VETS"), vets.size());
  }
}
```

PetClinic: JUnit 4.4 – TestContext POJO

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations="/clinic-context.xml")
@Transactional
public class PojoJUnit4ClinicTests {

  @Autowired
  protected Clinic clinic;

  protected SimpleJdbcTemplate simpleJdbcTemplate;

  @Autowired
  public void setDataSource(DataSource dataSource) {
    this.simpleJdbcTemplate = new SimpleJdbcTemplate(dataSource);
  }

  @org.junit.Test
  public void getVets() {
    Collection<Vet> vets = this.clinic.getVets();
    assertEquals("JDBC query must match",
      SimpleJdbcTestUtils.countRowsInTable(
        this.simpleJdbcTemplate, "VETS"), vets.size());
  }
}
```

Annotated POJO

Parameterized DI Test?

Q: Is it possible to use JUnit 4's **`Parameterized`** runner in conjunction with the dependency injection support in the Spring TestContext Framework?

A: Yes! As with the TestNG support, you can manually instrument your test with a **`TestContextManager`**, but… there are some limitations with this approach for JUnit 4.

Parameterized DI Test – Code (1)

```
@RunWith(Parameterized.class)
@ContextConfiguration
@TestExecutionListeners( {
  DependencyInjectionTestExecution
public class ParameterizedDependen

private final TestContextManager testContextManager;

@Autowired
private ApplicationContext applicationContext;

private final String employeeBeanName;
private final String employeeName;
```

Note: we cannot use both `Parameterized` and `SpringJUnit4ClassRunner` simultaneously!

Parameterized DI Test – Code (2)

```
public ParameterizedDependencyInjectionTests(
  String employeeBeanName, String employeeName)
  throws Exception {
    this.testContextManager =
      new TestContextManager(getClass());
    this.employeeBeanName = employeeBeanName;
    this.employeeName = employeeName;
}

@Parameters
public static Collection<String[]> employeeData() {
    return Arrays.asList(new String[][] {
      {"employee1", "John Smith" },
      {"employee2", "Jane Smith"} });
}
```

Parameterized DI Test – Code (3)


```
@Before
public void injectDependencies() throws Exception {
    this.testContextManager
      .prepareTestInstance(this);
}

@Test
public void testAgainstInjectedDependencies()
{
    // test against injected dependencies
    // with the 'parameterized‘ values
}
```

Demo: Parameterized DI Test

Custom TestExecutionListener?

Q: Is it possible to create a custom **`TestExecutionListener`** to provide project-specific testing support?

A: Yes! Either implement **`TestExecutionListener`** or extend **`AbstractTestExecutionListener`** and override the methods which suit your use case.

LogLevelTests (simple)

```
@RunWith(SpringJUnit4ClassRunner.class)
@TestExecutionListeners(
  LoggingTestExecutionListener.class)
public class LogLevelTests {

  @Test
  public void alwaysPasses() { /* no-op */ }

  @Test
  public void alwaysFails() {
    fail("always fails");
  }
}
```

LoggingTestExecutionListener (simple)


```
public class LoggingTestExecutionListener extends
    AbstractTestExecutionListener {

  public void prepareTestInstance(TestContext context)
      throws Exception {
    logInfo("Preparing test: " + context.getTestClass());
  }

  public void beforeTestMethod(TestContext context) {
    logInfo("Before method: " + context.getTestMethod());
  }

  public void afterTestMethod(TestContext context) {
    String msg = "After method: " + context.getTestMethod();
    if (context.getTestException() != null)
      logError(msg);
    else
      logInfo(msg);
  }
}
```

Demo: Simple Logging TestExecutionListener

Custom Test Annotations?

Q: Is it possible to provide support for custom annotations in tests?

A: Yes! Implement a custom **`TestExecutionListener`** which processes your custom annotations.

@LogLevel Custom Annotation

```
@Target({ElementType.TYPE,ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface LogLevel {

  public static enum Level {
     NONE, ERROR, WARN, INFO, DEBUG, TRACE;
  }

  Level value() default Level.ERROR;

}
```

LogLevelTests (annotation)


```
@RunWith(SpringJUnit4ClassRunner.class)
@TestExecutionListeners(
  LoggingTestExecutionListener.class)
@LogLevel(Level.NONE)
public class LogLevelTests {

  @Test
  public void alwaysPasses() { /* no-op */ }

  @Test
  @LogLevel(Level.INFO)
  public void alwaysFails() {
    fail("always fails");
  }
}
```

LoggingTestExecutionListener (annotation) (1)


```
public class LoggingTestExecutionListener extends
    AbstractTestExecutionListener {

  // modified prepare, before, and after methods

  private void logInfo(TestContext context, String msg) {
    if (getLogLevel(context).ordinal() >= INFO.ordinal()) {
      System.out.println("INFO: " + msg);
    }
  }

  private void logError(TestContext context, String msg) {
    if (getLogLevel(context).ordinal() >= ERROR.ordinal()) {
      System.err.println("ERROR: " + msg);
    }
  }

  // …
```

LoggingTestExecutionListener (annotation) (2)

```
private Level getLogLevel(TestContext context) {
  Level level = null;
  final Class<LogLevel> clazz = LogLevel.class;
  final Method testMethod = context.getTestMethod();

  if (testMethod != null)
    level = testMethod.isAnnotationPresent(clazz) ?
            testMethod.getAnnotation(clazz).value() : null;

  if (level == null) {
    final Class<?> testClass = context.getTestClass();
    level = testClass.isAnnotationPresent(clazz) ?
            testClass.getAnnotation(clazz).value() : null;
  }

  if (level == null)
    level = (Level) AnnotationUtils.getDefaultValue(clazz);

  return level;
}
```

Demo: @LogLevel-based Logging TestExecutionListener

PetClinic Sample Application

- Completely revised for Spring 2.5
- Annotation-driven configuration
- Annotation-driven @MVC controllers
  - focus on simple form handling
- Annotation-driven tests using the TestContext framework

Demo: PetClinic

Summary

The Spring TestContext Framework

- Provides generic unit and integration testing support
- Is agnostic of the actual testing framework
  - JUnit 3.8, JUnit 4.4, TestNG 5.5
- Embraces Java 5 annotations for configuration
- Provides a consistent yet enhanced feature set with an easy migration path for legacy code
- Is extensible, flexible, and easier to use

Further Resources

- Updated PetClinic sample application
- Spring test suite
- Spring Reference Manual
- Spring Forums
- blog.springsource.com
- www.springframework.org

Q&A

Sam Brannen
Senior Software Engineer
sam.brannen@springsource.com