

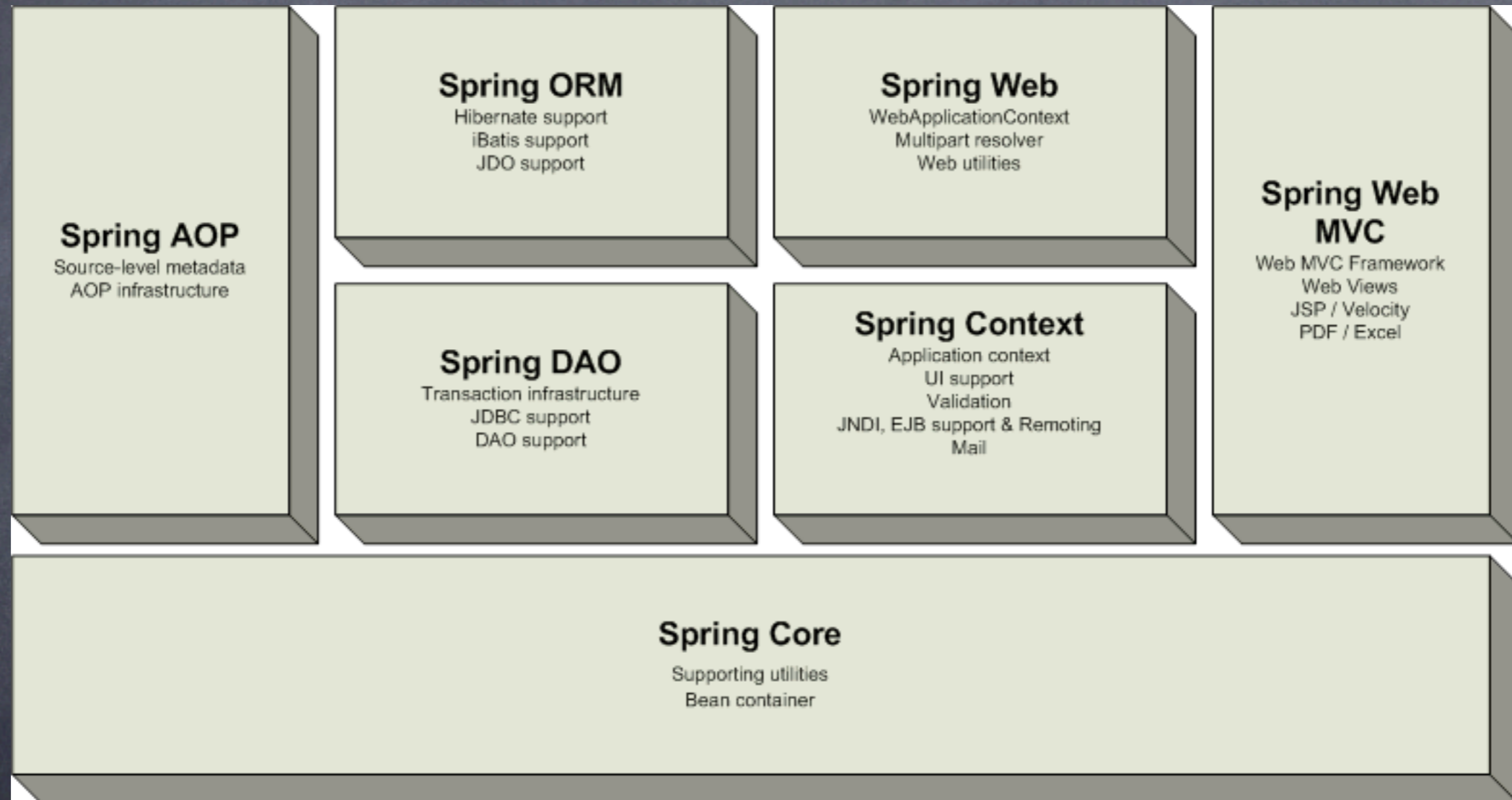
Facilitate TDD with the Spring Framework

By: Mike Shoemaker

What is Spring?

- Spring provides a light-weight solution for building enterprise-ready applications, while still supporting the possibility of using declarative transaction management, remote access to your logic using RMI or webservices, mailing facilities and various options in persisting your data to a database. Spring provides an MVC framework, transparent ways of integrating AOP into your software and a well-structured exception hierarchy including automatic mapping from proprietary exception hierarchies.

Spring Modules



Spring's Mission?

- J2EE should be easier to use
- It's best to program to interfaces, rather than classes. Spring reduces the complexity cost of using interfaces to zero.
- JavaBeans offer a great way of configuring applications.
- OO design is more important than any implementation technology, such as J2EE.
- Checked exceptions are overused in Java. A framework shouldn't force you to catch exceptions you're unlikely to be able to recover from.
- Testability is essential, and a framework such as Spring should help make your code easier to test.

Lightweight Container?

- Core usage does not impose unwanted compile time dependencies like EJB
- Loosely Coupled Components
- Unit testing is possible again
- POJO's

Role of the Container AKA BeanFactory

- Loads Bean definitions (XML)
- Instantiates Beans on behalf of client
- Configure dependencies via XML
- Replaces Factory methods for object creation

Singleton Beans

- Default configuration
- Bean instance is shared
- Lifecycle is managed by BeanFactory
- Efficient
- Good for Stateless operations

Prototype Beans

- BeanFactory offers a new instance upon each client request
- BeanFactory cannot manage lifecycle since it loses track of the bean after it hands it off
- Semantically equal to the "new" keyword

Sample Bean

```
<bean name="singletonBean"  
      class="com.acme.MyBean"  
      singleton="true" />
```

```
<bean name="prototypeBean"  
      class="com.acme.MyBean"  
      singleton="false" />
```

```
package com.acme;
```

```
public class MyBean {  
    private String state;
```

```
    public String getState() {  
        return state;  
    }
```

```
    public void setState(String state) {  
        this.state = state;  
    }  
}
```

How do I get a bean?

```
// String Array of bean definition file
```

```
String[] paths = {"applicationContext.xml"};
```

```
// Load bean definitions
```

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(paths);
```

```
// Code to retrieve bean from BeanFactory
```

```
MyBean bean = (MyBean)ctx.getBean("myBean");
```

ApplicationContext

- Extends BeanFactory functionality
- Can initialize singleton beans upon load
- Can lazy load beans
- Provides i18n support
- Supports lifecycle events

Implementations of ApplicationContext

- ClassPathXMLApplicationContext
- FileSystemXMLApplicationContext
- XmlWebApplicationContext
- Roll your own

Dependency Injection

- Synonymous with Inversion of Control (IOC)
- Sometimes referred to as JNDI in reverse
- "Hollywood Principal"
 - Don't call me, I'll call you
- DI coined by Martin Fowler

Types of Dependency Injection

- Interface Dependent (BAD)
 - Avalon
- Constructor Based
 - Pico Container
- Setter Based
 - Spring

Example of Constructor Based Injection

```
<bean id="dataSource" class="javax.sql.DataSource">  
  <property name="driverClassName">  
    <value>com.mysql.jdbc.Driver</value>  
  </property>  
  <property name="url">  
    <value>jdbc:mysql://localhost:3306/myDS</value>  
  </property>  
  <property name="username"><value>fred</value></property>  
  <property name="password"><value>password</value></property>  
</bean>
```

```
<bean id="loginDAO" class="com.acme.LoginDAOJDBC">  
  <constructor-arg>  
    <ref local="dataSource"/>  
  </constructor-arg>  
</bean>
```

Equivalent Java Code

```
Datasource datasource = .....
```

```
// Instantiate passing datasource in constructor
```

```
LoginDAO dao = new LoginDAOJDBC(datasource);
```

Example of Setter Based Injection

```
<bean id="dataSource" class="javax.sql.DataSource">  
  <property name="driverClassName">  
    <value>com.mysql.jdbc.Driver</value>  
  </property>  
  <property name="url">  
    <value>jdbc:mysql://localhost:3306/myDS</value>  
  </property>  
  <property name="username"><value>fred</value></property>  
  <property name="password"><value>password</value></property>  
</bean>
```

```
<bean id="loginDAO" class="com.acme.LoginDAOJDBC">  
  <property name="dataSource">  
    <ref local="dataSource"/>  
  </property>  
</bean>
```

Equivalent Java Code

```
Datasource datasource = .....
```

```
// Instantiate
```

```
LoginDAO dao = new LoginDAOJDBC();
```

```
// set data source that was acquired previously
```

```
dao.setDatasource(datasource);
```

References

- <http://www.springframework.org>
- Spring Live
- Spring In Action
- Pro Spring

Group Exercise

- ① Create Car Insurance Application without Spring
- ① Once it's working, introduce Spring
- ① Discuss Other Possibilities