

Dependency Injection and Spring

Rod Johnson
Interface21

Topics

- Dependency Injection
 - What?
 - Why?
 - Spring configuration concepts and examples
- Questions/Discussion

Topics

- Dependency Injection
 - Examples
 - Types of dependency injection
 - Autowiring
 - The real world: Beyond the basics

Let's begin with an example...

Middle Tier: Setter Injection

```
public class ServiceImpl implements Service {
    private int timeout;
    private AccountDao accountDao;

    public void setTimeout(int timeout) {
        this.timeout = timeout;
    }

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    // Business methods from Service
    ...

<bean id="service" class="com.mycompany.service.ServiceImpl">
    <property name="timeout"><value>30</value></property>
    <property name="accountDao"><ref local="accountDao"/></property>
</bean>
```

Middle Tier: Constructor Injection

```
public class ServiceImpl implements Service {
    private int timeout;
    private AccountDao accountDao;

    public ServiceImpl (int timeout, AccountDao accountDao) {
        this.timeout = timeout;
        this.accountDao = accountDao;
    }

    // Business methods from Service

<bean id="service" class="com.mycompany.service.ServiceImpl">
    <constructor-arg><value>30</value></constructor-arg>
    <constructor-arg><ref local="accountDao"/></constructor-arg>
</bean>
```

Traditional approach

- Hard-code use of *new*
 - What if something changes?
 - How do we externalize configuration from Java code, important if things change
- Use a custom factory
 - More code to write in the application
 - Just move the hard-coding or ad-hoc parameterization one step farther away
- ... “Service Locator” approach traditional in J2EE

Benefits of Dependency Injection

- Unit testable
- Dependencies are explicit
- Consistent
- Can wire up arbitrarily complicated graphs
- You don't need to write plumbing code
- Pluggability
 - Reduces cost of programming to interfaces to zero

*Spring provides a factory to end all
factories*

Unit testing

```
public void testFindAccounts() {  
  
    Customer cust = new Customer();  
  
    MockControl mc = MockControl.createControl(AccountDao.class);  
    AccountDao mockDao = (AccountDao) mc.getMock();  
    mockDao.findAccounts(cust);  
    mc.setReturnValue(new LinkedList());  
    mc.replay();  
  
    Service s = new ServiceImpl();  
    s.setAccountDao(mockDao);  
  
    assertTrue("No accounts to calculate",  
               0, s.calculateInterestPayable(cust));  
    mc.verify();  
}
```

Middle Tier: Autowiring

```
public class ServiceImpl implements Service {
    private int timeout;
    private AccountDao accountDao;

    public ServiceImpl (int timeout, AccountDao accountDao)
    {
        this.timeout = timeout;
        this.accountDao = accountDao;
    }

    // Business methods from Service

<bean id="service"
    class="com.mycompany.service.ServiceImpl"
    autowire="byProperty" />
```

Types of autowiring

- Autowiring applies to both Setters and Constructors
 - Sophisticated algorithm to resolve “greediest” constructor
- Autowiring by type
 - Resolve a dependency by its type, where there is exactly one candidate managed object
- Autowiring by name
 - Look for a managed object with the same service name as the property

Autowiring: Advantages

- Reduces volume of configuration
- Configuration can “learn” about your code
 - If you add a property, it may automatically be populated
 - Particularly useful during periods of rapid change in code, such as prototyping and early development

Autowiring: Disadvantages

- More magical
- Less self documenting
- Less amenable to analysis by tools that generate documentation or graphics from Spring configs

Spring config: Simple property

```
<property name="foo">  
  <value>literalValue</value>  
</property>
```

Spring config: Reference

```
<property name="fooHelper">  
  <ref bean="fooHelper"</ref>  
</property>
```

Spring config: "Inner bean"

```
<property name="fooHelper">  
  <bean class="package.FooHelperImpl">  
    <property...  
  </bean>  
</property>
```

Spring config: List

```
<property name="things">
  <list>
    <ref bean="thing1" />
    <ref bean="thing2" />
    <value>literalValue</value>
  </list>
</property>
```

Advanced features

- Can't always use the constructor you want
 - May work with legacy objects
- Factory beans
- Lifecycle callbacks
- Sometimes an API is required
- Spring supports all forms of Inversion of Control

Construction styles

```
<bean class="FooImpl"  
      factory-method="createFoo" />
```

```
<property...>
```

```
</bean>
```

Instantiation choices

- **Constructor**
 - With or without arguments, which can be other definitions or literals
- **Static method**
 - Can have arguments, which can be other definitions or literals
- **Method on another bean in the context**
 - Mainly intended for use within Spring framework code, but may be useful for particularly advanced configuration scenarios in Spring applications
- **“Factory Bean”**
 - Class with special meaning to the framework, introducing a level of indirection

JndiObjectFactoryBean

```
<bean id="dataSource"  
    class="org.springframework.jndi.JndiObjectFactoryBean"  
    >  
    <property name="jndiName">  
        <value>oracle</value>  
    </property>  
</bean>
```

```
<bean id="accountDao"  
    class="com.mycompany.JdbcAccountDao">  
  
    <property name="dataSource">  
        <bean local="dataSource"/>  
    </property>  
</bean>
```

Comparison: Local datasource

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"  
    destroy-method="close">  
    <property name="driverClassName">  
        <value>${jdbc.driverClassName}</value>  
    </property>  
    <property name="url"><value>${jdbc.url}</value></property>  
    <property name="username"><value>${jdbc.username}</value></property>  
    <property name="password"><value>${jdbc.password}</value></property>  
    <property name="maxActive"><value>30</value></property>  
    <property name="defaultAutoCommit"><value>>false</value></property>  
    <property name="poolPreparedStatements"><value>>true</value></property>  
</bean>
```

Notes on the last slide

- Change only the “dataSource” bean definition
- Illustrates how properties can be externalized to properties files
 - Intended for maintenance of key values without touching Java *or* XML
- Shows use of “init-method” and “destroy-method”
 - Commons DBCP BasicDataSource has no knowledge of Spring, but Spring can manage its lifecycle as well as set properties

FactoryBean: General mechanism for aliasing...

- Switch between POJO and remote service...
- Switch between POJO and EJB...
- Add custom behaviour
- Don't normally need to *implement* factory beans (come with framework) but it's important to know how they work

Accessing an EJB

```
<bean id="myComponent"  
  class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">  
  <property name="jndiName"><value>myComponent</value></property>  
  <property name="businessInterface">  
    <value>com.mycom.MyComponent</value>  
  </property>  
</bean>
```

- No Service Locator or Business Delegate
- *No Java code at all*
- Spring performs JNDI lookup, calls EJBHome.create()
- Same approach works for web services etc

Factory bean interface

```
public interface
    org.springframework.beans.factory.FactoryBean {

    Object getObject() throws Exception;

    Class getObjectType();

    boolean isSingleton();

}
```

- Factory beans are themselves be configured by DI...

Lifecycle callbacks

- BeanFactoryAware
- ApplicationContextAware
- InitializingBean
- DisposableBean

```
public void setBeanFactory(BeanFactory bf) {  
}
```

Lifecycle callbacks

- If you want a lifecycle callback, you implement just that interface
- The container automatically recognizes this and makes the necessary callback
- Can also specify “init” and “destroy” methods, as shown with `BasicDataSource` on previous slide
 - Any no-arg method
 - Allows existing classes that need explicit initialization or cleanup to be fully configured by Spring, with no Spring dependencies

API

- BeanFactory
- ApplicationContext
- ListableBeanFactory
- Permits explicit lookup

```
public Object getBean(String name) ;
```

When to use lookup?

- Most of the time (probably around 95%) you will *not* need to use Spring container APIs
- Examples of valid use of container APIs:
 - You need to look for a particular object at runtime
 - You need to look for a particular object in the context of a particular operation

Spring, DI and performance

- By default, Spring simply instantiates and “wires” your POJOs
- Spring container produces *no* performance overhead
- Proxying behaviour is added only if you specify it
- No limits have emerged regarding the size of Spring contexts

Manageability

- Containers can participate in hierarchies
- Can combine multiple XML files to build one context
 - No need to have one monolithic XML file
- Can capture common functionality in common “template” definitions

Template definitions

```
<bean id="txTemplate"  
  class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">  
  <property name="transactionManager">  
    <ref local="transactionManager"/>  
  </property>  
  
  <property name="transactionAttributes">  
    <props>  
      <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>  
      <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>  
    </props>  
  </property>  
</bean>
```

```
<bean id="clinic" parent="txTemplate">  
  <property name="target"><ref local="clinicTarget"/></property>  
</bean>
```

- Common properties are “inherited”
- Can use to apply consistent approach across many definitions

Object lifecycle

- Default is for a bean to have a single, shared, instance in a context
 - All references will be to this object
 - All `getBean()` calls will be returned this object
- Can specify `singleton="false"` to ensure that
 - References are to independent instances
 - All `getBean()` calls get distinct objects

Non-singleton usage and pooling

- Allows for single-use objects
 - Combine with “Method Injection” to get single-use objects without an explicit lookup
- Shared instance model is most appropriate in general
- Spring also supports instance pooling for POJOs (similar to SLSB instance pooling), but pooling of object instances is not usually necessary
 - Pooling of *resources* is important

Unit testing

- *Unit testing* doesn't need to consider Spring at all in most cases
 - The point of Spring is to conceal infrastructure

Integration testing

- **Extend** `org.springframework.test` superclass
- **Test populated by dependency injection**
 - No need for explicit lookup
- **Superclasses also provide**
 - Caching of contexts for all test cases
 - Transactional behaviour