ENTERPRISE DEVELOPMENT SERVICES DIRECT FROM THE CREATORS OF SPRING FRAMEWORK

INTERFACE21

Configuring the Spring Container

Rod Johnson

CEO

Interface21

Spring

Topics

- **Spring container philosophy**
- Spring configuration metadata
- XML configuration
- XML alternatives
  - Spring 2.5 annotations
  - Spring Java Config
  - Scripting language configuration
- Recommendations

Spring Container Philosophy

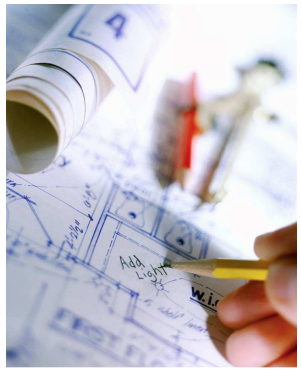
- Mission
  - To provide the ultimate component model for the enterprise
  - To support different programming models on a common foundation
  - To provide value adds for components, however defined
    - True AOP
    - Transaction management
    - JMX
    - Third party integrations…
- Expressed through a *contributions* approach

Spring

Topics

- Spring container philosophy
- **Spring configuration metadata**
- XML configuration
- XML alternatives
  - Spring 2.5 annotations
  - Spring Java Config
  - Scripting language configuration
- Recommendations

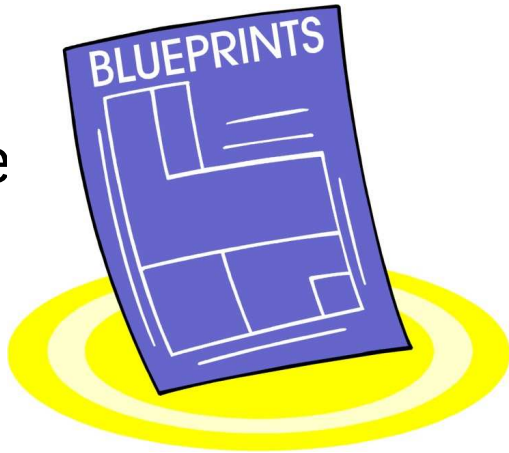
Spring Configuration

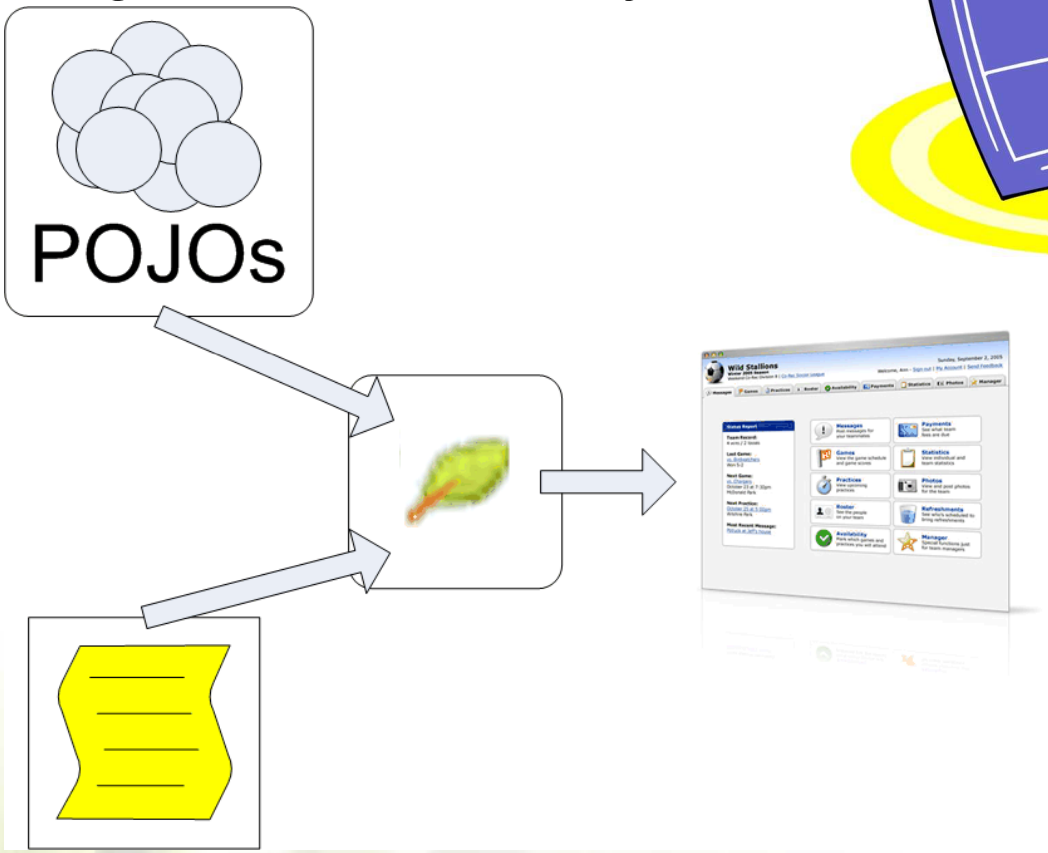


- Java metadata used internally by the container for
  - Instantiation
  - Configuration
  - Decoration
  - Assembly
  - Instance Management (lifecycle)

In short

- Spring + cfg + classes = ready-to-use





The BeanDefinition interface

- Contains
  - Bean class or parent
  - Properties
  - Constructor args
  - Scope
  - “Autowiring” information
  - …More advanced stuff
- The XML <bean> element carries the same metadata

Using the XML <beans> element

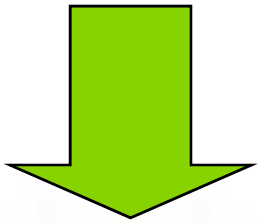
```
<beans>
 <bean id="accountService" class="...DefaultAccountService">
    <property name="accountDAO" ref="accountDao" />
  </bean>

  <bean id="accountDAO" class="...JdbcAccountDAO"
                        init-method="init">
    <property name="dataSource" ref="dataSource" />
  </bean>

</beans>
```

XML Namespaces: Since Spring 2.0

- Key benefit:
  - Allows a higher level of abstraction
    - Better express the intent
    - More concise
- No longer a 1:1 mapping between XML element and bean definition
- Can emit 0 or more bean definitions

Using a namespace

```
<bean id="dataSource“
                  class="...JndiObjectFactoryBean">
  <property name="jndiName" value="jdbc/AccountData"/>
</bean>
```



```
<jndi:lookup id="dataSource"
                        jndiName="jdbc/AccountData"/>
```

When to use XML namespaces

- To define beans of the same class repeatedly, and set the same properties each time
- To define a group of beans that must work together
- To create a configuration DSL that will be reused across a project or company.
- For conditional contribution that may generate *no* bean definitions in some cases, or even alter other bean definitions
- To create an abstraction between configuration file and implementing class
- To migrate existing XML formats to configure Spring

When *not* to use XML namespaces

- To define application classes that will be used only once or very few times
- Remember the lessons of JSP custom tags??
- Remember
  - <bean> definitions are universally understood

Summary: XML Pros

- **Most powerful configuration option**
  - **Offers per-instance control, which you can't get with annotations**
- Easy to understand
- Good for defining simple type values
- Externalized from code
  - Can change configuration without recompilation
  - Configuration can be changed by non Java developers
- Platform independent
- Supports validation
  - Especially with schemas
- Excellent IDE support with Spring IDE, IntelliJ

XML Cons

- Refactoring unfriendly
- Reliance on String identifiers
- Verbose
  - Angle bracket noise
- Limited hierarchical model
- Files can become large

Topics

- Spring container philosophy
- Spring configuration metadata
- XML configuration
- **XML alternatives**
  - **Spring 2.5 annotations**
  - **Spring Java Config**
  - **Scripting language configuration**
- Recommendations

Annotations

- Add metadata to source code
- Spring has offered annotations for enterprise services (such as @Transactional) since 1.2
- Comprehensive annotation support for DI introduced in Spring 2.5

Purpose of annotations for DI

- **Annotations applied to classes, methods or fields**

- Annotations on *classes* identify components to be managed by Spring
- Annotations on *methods* identify methods whose arguments should be injected
  - Can have multiple arguments
- Optional annotations on *method arguments* provide information about how to resolve dependency
- Annotations on *fields* identify value that should be injected

Spring *stereotype* annotations

- @Service
  - Identifies a stateless service
- @Repository
  - Identifies a repository (DAO)
- @Aspect
  - @AspectJ aspect
- @Controller
  - Spring MVC controller

- Can define your own…
- @Component
  - *Meta-annotation*
    - Annotate your own annotation with @Component and hey presto! your classes get picked up by scanning

A word on appropriate use of annotations

- Annotations are ideal for indicating the *role* of something in an application
- *Not* ideal for carrying string values or other implementation-specific details
  - Does not result in strong typing
  - Means Java code needs to be recompiled to change values that should be externalized

- **Stereotypes are an ideal use**

Component Scanning

- Scans the classpath for annotated classes
- Removes the need for XML definitions unless you want to do something you can't do in annotations

```
@Service
public class DefaultAccountService { ...
```

```
<bean id="defaultAccountService"
                      class="DefaultAccountService"/>
```

Component Scan Usage

- Use *context* namespace
- Specify package to pick up
- Can coexist with XML bean definitions and namespaces

```
<context:component-scan
      base-package="com.mycompany.myapp"/>
```

More advanced component scanning usage

- Not limited to annotations
  - Can use type or other checks
- Highly customizable, as you expect from Spring

```
<context:component-scan base-package="blog"
                        use-default-filters="false">
    <context:include-filter type="annotation"
       expression="org.sf.stereotype.Component"/>
    <context:include-filter type="regex"
       expression="blog\.Stub.*"/>
    <context:exclude-filter type="assignable"
       expression="blog.JdbcMessageRepository"/>
</context:component-scan>
```

Component Scan Pros

- No need for XML unless you need the greater sophistication it allows
- Changes are picked up automatically
  - Great during development
- Works great with Annotation Driven Injection
  - picking up further dependencies with @Autowired
- Highly configurable

Component Scan Cons

- Not a 100% solution
  - Can't do everything with annotations
- Requires classes to be annotated
- Need to take care not to scan an excessive number of classes, using Spring's filtering mechanism
- Don't get the valuable application structure blueprints you get with XML configuration
  - Although Spring IDE can unify all Spring component definitions

Resolving Dependencies: @Autowired

- Provides injection at constructor/field/method level
- Supports multi argument methods

```
@Autowired
public void createTemplates(DataSource ds,
                            ConnectionFactory cf) {
  this.jdbcTemplate = new JdbcTemplate(ds);
  this.jmsTemplate = new JmsTemplate(cf);
}
```

Resolution of dependencies by name

```
public class JdbcOrderRepositoryImpl
   implements OrderRepository {

   @Autowired
   public void init(
       @Qualifier("myDataSource“) orderDataSource,
       @Qualifier("otherDataSource")
       inventoryDataSource,
       MyHelper autowiredByType) {
             // ...
}
```

Resolution of dependencies by annotation

```
public class JdbcOrderRepositoryImpl
  implements OrderRepository {

  @Autowired
  public void setOrderServices(
      @Emea OrderService emea,
      @Apac OrderService apac) {
          // ...
}
```

JSR-250 annotations also supported

- `@PostConstruct`
  - Similar to `InitializingBean#afterPropertiesSet()`
- `@PreDestroy`
  - Similar to `DisposableBean#destroy()`
- `@Resource`
  - Identifies injection point

@Resource Example

```
public class DefaultAccountService
                    implements AccountService {

   @Resource
   private AccountDAO jdbcAccountDAO;
   ...
}


public class JdbcAccountDAO implements AccountDAO {

   @PostConstruct
   public void init() {...} ...
}
```

@Resource Pros

- Supports Java EE 5 configuration style
  - Note: Spring does *not* require that dependencies are resolved from JNDI, although it supports this
- Compiler support through annotations
- Reuses annotation context
- Fine grained injection

@Resource Cons

- Classes need to be annotated
- Unsophisticated
  - @Resource style is not as powerful as Spring @Autowired approach
  - No support for “qualifiers” or annotation resolution

ENTERPRISE DEVELOPMENT SERVICES DIRECT FROM THE CREATORS OF SPRING FRAMEWORK

INTERFACE21

Spring Java Configuration

Spring

Spring Java Configuration

- Annotation-centric approach, but very different
  - Annotations are in dedicated configuration classes, *not* application classes
- Allows objects to be created and wired in Java

@Configuration

- Similar to <beans/>
- Specifies a configuration class
- Defines defaults for the current context

```
@Configuration(
   defaultAutowire = Autowire.BY_TYPE,
   defaultLazy = Lazy.TRUE)
```

@Bean

- Similar to <bean>
- Indicates a bean creation method
- Supports standard bean attributes from BeanDefinition internal metadata
  - lazy
  - scope
  - depends-on

@Bean

```
@Bean (scope = REQUEST)
public Page currentPage() { … }


@Bean (scope = SESSION,
       destroyMethodName = “shutdown”);
public Preferences prefs() { … }


@Bean (lazy = Lazy.FALSE);
public Admin admin() { … }
```

Java Configuration Class Example

```
@Configuration
public abstract class JavaConfig {
  @Bean
  public AccountDAO accountDAO() {
      // return new InMemoryAccountDAO();
      JdbcAccountDAO dao = new JdbcAccountDAO();
      dao.setDataSource(dataSource());
      dao.init();
      return dao;
  }
  @Bean
  public AccountService accountService() {
      DefaultAccountService service = new DefaultAccountService();
      service.setAccountDAO(accountDAO());
      return service;
  }

  @ExternalBean
  public abstract DataSource dataSource();
}
```

Bean-to-Bean Dependencies are handled elegantly

```
@Bean
public AccountDAO accountDAO() { ... }
...
service.setAccountDAO(accountDAO());
```

```
service.setAccountDAO(
                ctx.getBean("accountDAO"));
```

@ExternalBean – Reference external beans

- Easy way to reference external beans
- Strongly typed

```
@ExternalBean
public abstract DataSource dataSource();
```

```
public DataSource dataSource() {
     return (DataSource) ctx.getBean("dataSource");
}
```

@ExternalValue – Reference external properties

- Easy way to reference external property values
- Strongly typed

```
@ExternalValue
public abstract int getAge();
```

```
public int getAge() {
     // Look up external properties value and return
}
```

Private/Hidden beans

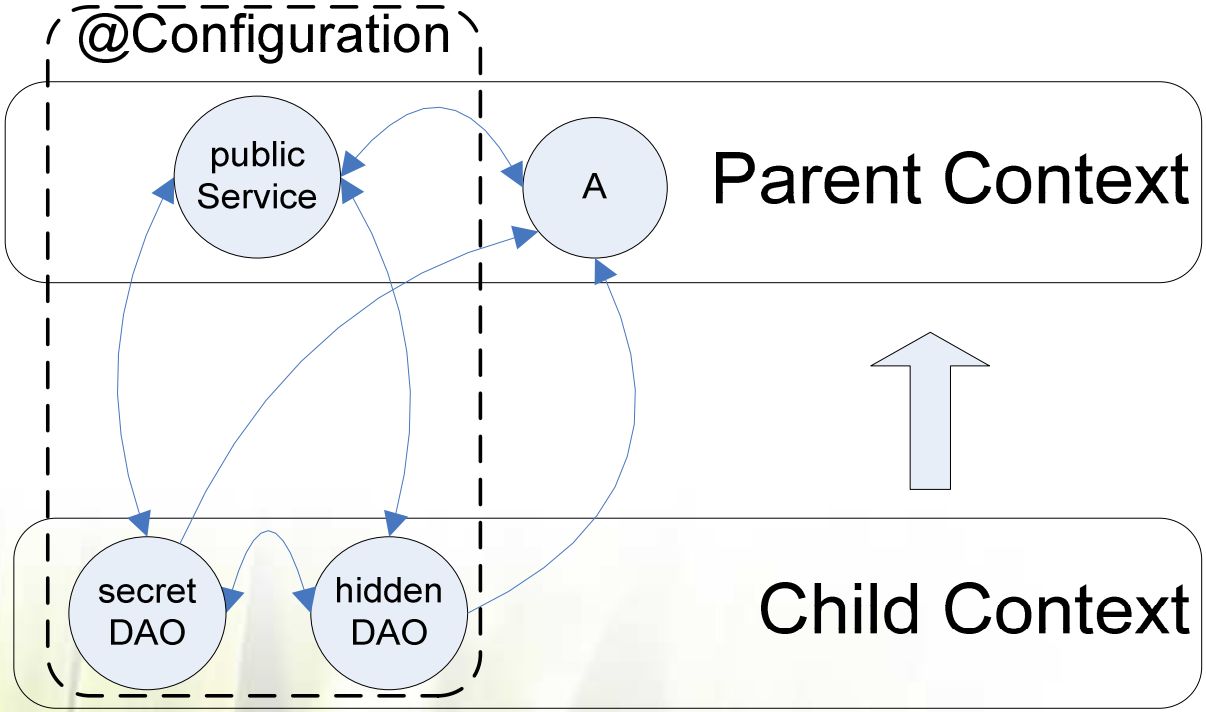
- Unique feature

- Non-public methods create ‘private’ beans
- Invisible to the ‘owning’ context
- Similar to inner beans but with full scope support
- Visible only to beans inside the same configuration

Private beans example

```
@Bean
public AccountService accountService() {
  AccountService service = new DefaultAccountService();
  service.setAccountDAO(hiddenDAO());
   return service;
}

@Bean
protected AccountDAO hiddenDAO() {
   return new InMemoryAccountDAO(); }

@Bean
private Object secretDAO() {
   return new JdbcAccountDAO();}
```

Private beans and contexts



Bootstrapping Java Configuration

- Dedicated application context

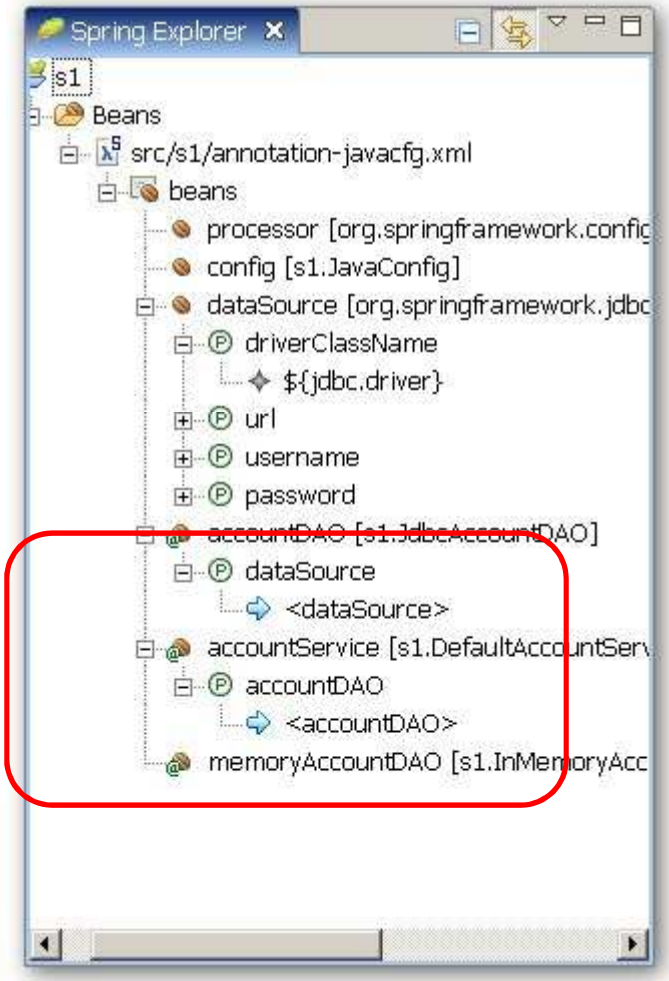
```
ApplicationContext ctx =
new AnnotationApplicationContext(
                "**/springone/JavaConfig.class");
```

- BeanFactoryPostProcessor allows use in a regular application context
- Just define @Configuration classes as beans
  - Can inject them normally

```
<bean class="org.springone.JavaConfig"/>
<bean class="org.springframework.config.java.
           process.ConfigurationPostProcessor"/>
```

Java Configuration Pros

- Pure Java
  - Allows visibility control
  - Allows use of inheritance in configurations
- Powerful object creation
  - Ability to use arbitrary Java code
  - Good for configuring existing classes
- Refactoring friendly
- Strongly typed
- Preserves valuable application blueprint
- IDE support with Spring IDE



Java Configuration Cons

- Configuration changes require recompilation
- Requires CGLIB
  - no final classes/methods
  - only for configuration classes

Annotation configuration vs Spring Java Configuration

- Different philosophies
  - Annotation driven injection *adds metadata to container identifying components and injection methods*
  - Java Configuration is *programmatic object creation*

Mix and Match

- All Spring metadata in the end
- One approach does not exclude others
- Can have multiple contributions to the one context

Example: Spring JavaConfig + XML

```
@Bean
EditOwnerForm editOwnerForm() {
   EditOwnerForm form = new EditOwnerForm()

   form.formView = "ownerForm"
   form.successView = "ownerRedirect"
   form.validator = ownerValidator()
   form.clinic = clinic()

   return form
}

// defined through XML
@ExternalBean
abstract Clinic clinic()
```

```
<bean id=“clinic"
  class=“...JpaClinic">
...
</bean>
```

Spring IDE Visualization and Editing support

- Spring IDE provides sophisticated visualization and editing support for bean definitions, *however defined*
- Unified view of configuration

Configuration overview

- Configuration becomes more static over time
  - Except for simple configuration properties, which should be externalized from Java code
- Static wiring
  - Java Configuration + Annotation DI
- Not so static (changes all the time) configuration
  - XML
- Simple values (urls and passwords)
  - Properties files, externalized from XML or Java
- Specialized configurations
  - DSL / XML namespaces

Future Directions

- Will continue to offer additional configuration options for our strong, extensible component model
- May also offer dynamic configuration
  - Database
  - “Warm” and “cold” start

Summary

- Spring > XML
  - Provides the ultimate component model for enterprise Java
- Often appropriate to use more than one strategy
  - Can mix and match
  - Choose the best approach *for each requirement*

Spring is About Choice

- Be Pragmatic
- Be Consistent

- Unlike other solutions, Spring does not aim to impose behaviors
  - No one size fits all
  - This is one of the secrets of Spring's success…

Q&A