

An XML-Free Spring

By Chris Hardin

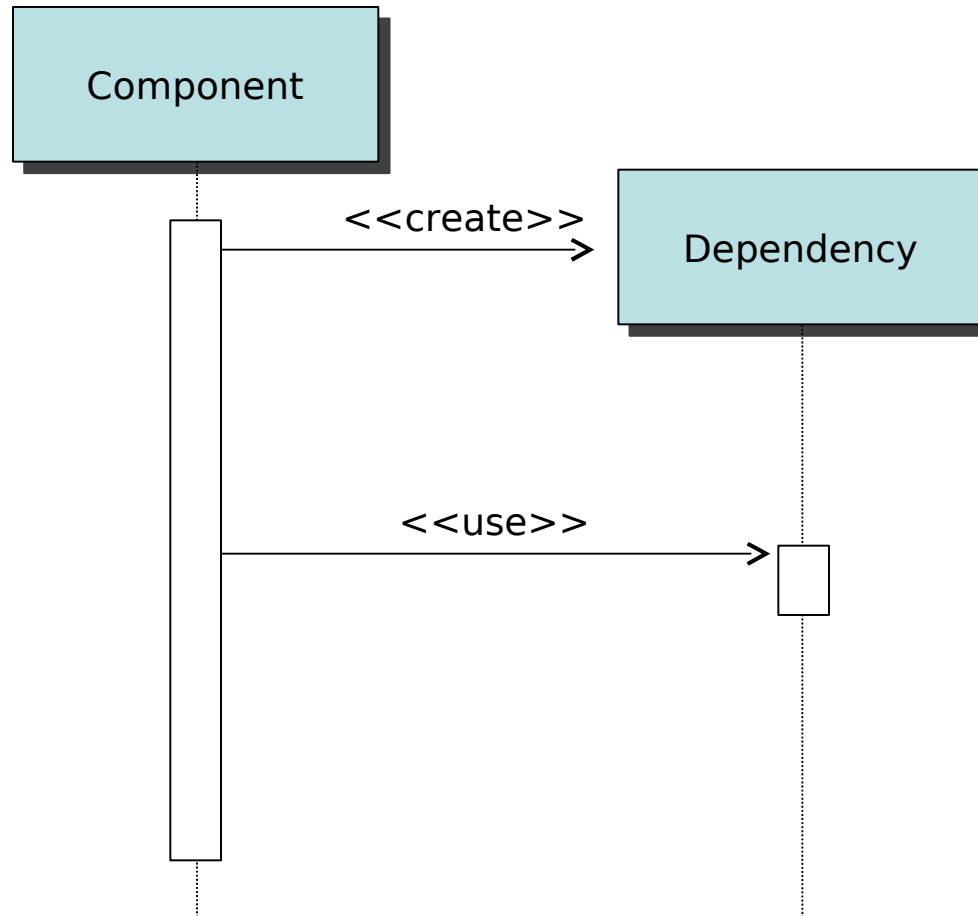
What is Spring?

- Spring is a **framework** that provides building blocks for common application needs
- Spring is an Inversion of Control **container** that creates the components of your application and manages their life cycles and relationships

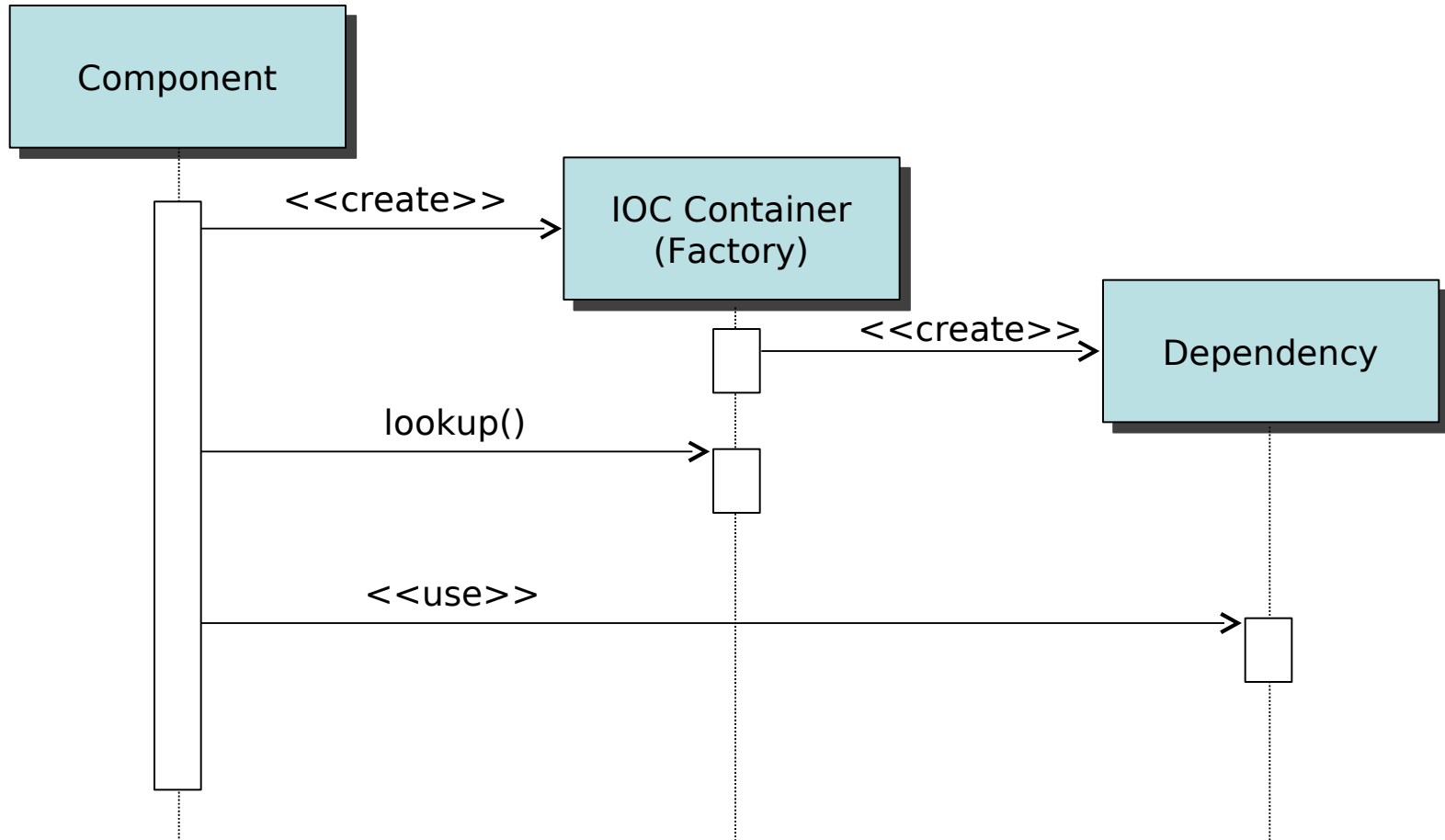
Inversion Of Control

- IOC is the idea that components should not have control over the life cycle of other objects on which they depend
- IOC is the core idea behind Spring
- Two implementation styles:
 - Dependency Lookup
 - Provides facility for components to locate instances of the dependencies they need
 - **Dependency Injection**
 - Injects instances of dependencies into the component using setters or constructor arguments

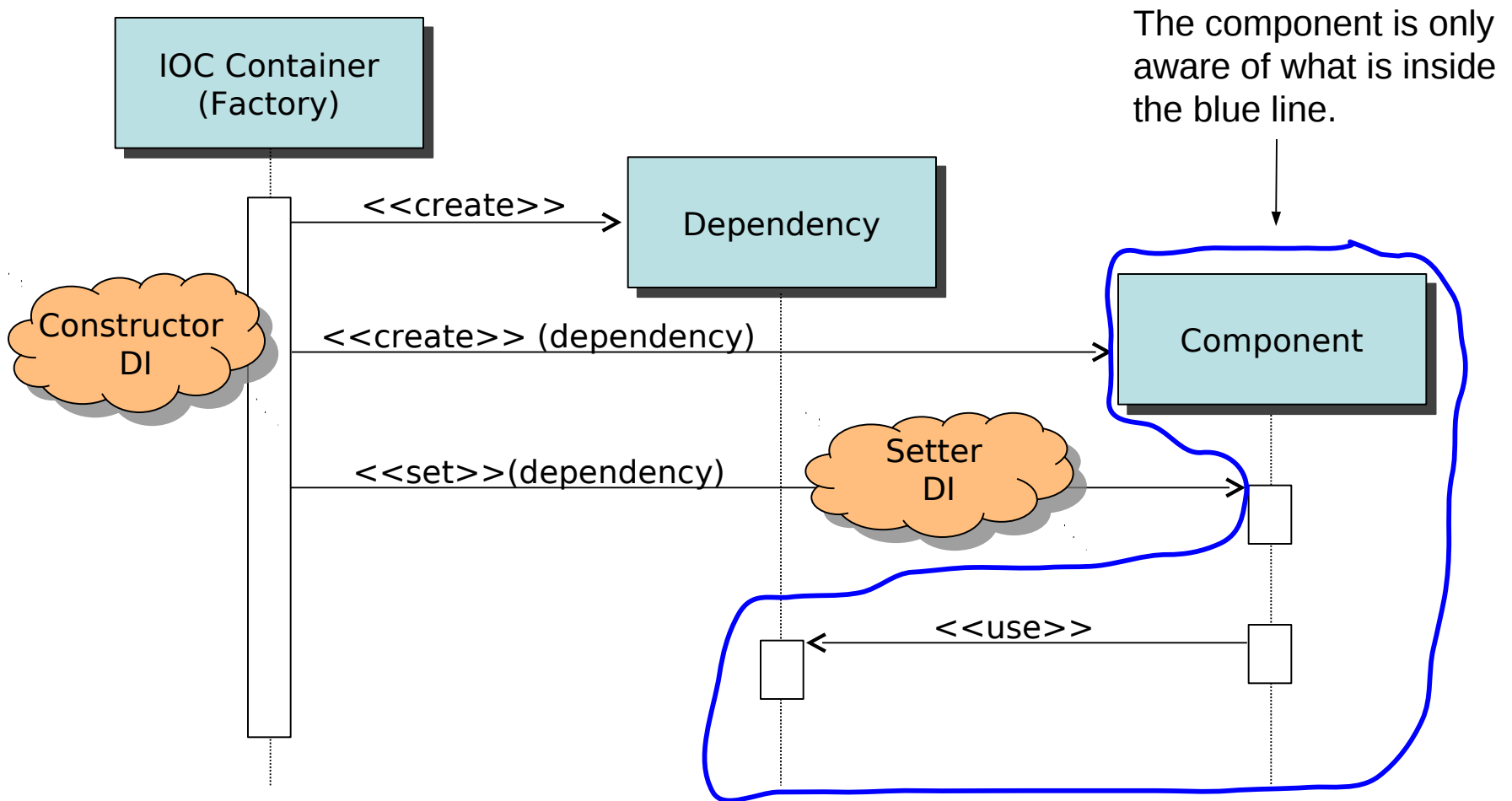
Normal Flow of Control



IOC: Dependency Lookup



IOC: Dependency Injection



Benefits of Using an IOC Container

- Reduce glue code
- Externalize dependencies
- Manage dependencies in a single place
- Improve testability
- Encourages good design using interfaces
- Change implementations without modifying existing **application** code

Injection vs. Lookup

- Choice mandated by the container
- Spring primarily uses injection
- Injection allows components to be reused in any environment or container
- Lookup couples components to the container
- Lookup is more complex, more code
- Testing is easier with injection

Spring IOC Implementation

- A BeanFactory manages components and their dependencies (beans)
- Bean configuration is stored inside the factory using BeanDefinition objects
- Use a BeanDefinitionReader to load the configuration into the factory

Bean Configuration Metadata

- XML
 - Oldest type of configuration
 - All beans are defined in the XML file
- Java Annotations
 - Added fully in Spring 2.5
 - Annotations are added directly to your **application** classes
- **Java Code**
 - New in Spring 3.0
 - Special Java classes contain bean configuration separate from your application code
 - These configuration classes act like internal bean factories

Java Configuration Classes

- Spring configuration annotations let you replace XML bean definitions with Java code
- These annotated configuration classes can be used alone or in combination with XML and annotated application classes
- As of Spring 3.0, they only support simple bean definitions, the various XML namespaces are still useful for more complex configuration
- Spring 3.1 will expand the capabilities of the Java-based configuration

Traditional XML Configuration

```
<bean id="greetingService" class="com.example.GreetingService"  
      c:_0-ref="source" c:_1-ref="destination" />
```

← C namespace
added in 3.1

```
<bean id="source" class="com.example.GreetingSourceImpl"  
      p:greeting="Hello World" />
```

```
<bean id="destination" class="com.example.GreetingDestinationImpl"/>
```

Annotated Application Class

@Component

```
public class GreetingService implements Service {
```

```
    private GreetingSource source;  
    private GreetingDestination destination;
```

@Autowired

```
public GreetingService(GreetingSource source, GreetingDestination destination){  
    this.source = source;  
    this.destination = destination;  
}
```

```
public void execute() {  
    destination.write(source.getGreeting());  
}
```

```
}
```

What implementations of source and destination
will this class get injected with?

Where do you look to find out?

How do you change the configuration?

Using an Annotated Configuration Class

(Back to the Future?)

```
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;
```

@Configuration

```
public class BeanConfig {
```

@Bean

```
public GreetingSource source(){  
    GreetingSourceImpl s = new GreetingSourceImpl();  
    s.setGreeting("Hello World");  
    return s;  
}
```

Method name becomes the bean name



@Bean

```
public GreetingDestination destination(){  
    return new GreetingDestinationImpl();  
}
```

An @Bean
annotated method
replaces each
<bean /> tag in
the old XML

@Bean

```
public Service greetingService(){  
    return new GreetingService(source(), destination());  
}
```

Call other @Bean methods to inject dependencies



Review Goals of an IOC Container

- Reduce glue code
- Externalize dependencies
- Manage dependencies in a single place
- Improve testability
- Encourages good design using interfaces
- Change implementations without modifying existing **application** code

Advantages of Java-Based Configuration

- Compile-time checking—no more XML typos discovered at runtime!
- Full IDE support without special tools or plugins
- Easy traceability
- No special syntax/tags to remember, just write plain old Java
- You can **do whatever you want** in an @Bean annotated method, such as:
 - Call utility methods to avoid code duplication
 - Use other types of builders or factories
 - Pull values from properties or other resources

Bean Instantiation Modes

```
<bean id="destination" class="com.example.GreetingDestinationImpl"  
      scope="singleton"/>
```

- singleton - one instance, default
- prototype - everyone gets new instance
- request - one per HTTP request
- session - one per HTTP session
- globalsession - one per global HTTP session

Using @Bean Attributes and @Scope

```
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Scope;
```

```
@Configuration
```

```
public class BeanConfig {
```

```
    @Bean(name = {"source", "greetingSource", "bob"},  
          destroyMethod="destroy")
```

← Bean Aliases

```
    public GreetingSource source(){  
        GreetingSourceImpl s = new GreetingSourceImpl();  
        s.setGreeting("Hello World");  
        s.init();  
        return s;  
    }
```

← Can call lifecycle methods directly

```
    @Bean(name="consoleDestination",  
          initMethod="init")
```

```
    @Scope("prototype")
```

```
    public GreetingDestination destination(){  
        return new GreetingDestinationImpl();  
    }
```

← Specify name, lifecycle
callbacks, and scope

```
    @Bean
```

```
    public Service greetingService(){  
        return new GreetingService(source(), destination());  
    }
```

```
}
```

How Does it Respect Bean Scopes?

- If you have a bean with singleton scope, you will always get the same instance when you call the @Bean annotated method that returns it.
- Spring creates a CGLIB proxy of your configuration class. Every time an @Bean method is called, the proxy checks for a cached instance of the object and returns it if appropriate or then calls your method implementation to get a new object.
- Configuration classes must have a no-arg constructor and must not be `final`

@Lazy, @Primary, @DependsOn

```
import org.springframework.context.annotation.DependsOn;
import org.springframework.context.annotation.Lazy;
import org.springframework.context.annotation.Primary;

@Configuration
public class BeanConfig {

    @Bean(name = {"source", "greetingSource", "bob"},
          destroyMethod="destroy")
    public GreetingSource source(){
        GreetingSourceImpl s = new GreetingSourceImpl();
        s.setGreeting("Hello World");
        s.init();
        return s;
    }

    @Bean(name="consoleDestination",
          initMethod="init")
    @Scope("prototype")
    @Primary
    @DependsOn("someBean")
    public GreetingDestination destination(){
        return new GreetingDestinationImpl();
    }

    @Bean
    @Lazy
    public Service greetingService(){
        return new GreetingService(source(), destination());
    }
}
```

Loading the Application Context

```
public class HelloWorld {  
    public static void main(String[] args) {  
        ApplicationContext ac =  
            new AnnotationConfigApplicationContext(BeanConfig.class);  
        Service service = (Service) ac.getBean("greetingService");  
        service.execute();  
    }  
}
```

Pass in your configuration classes to the application context constructor and it loads your beans without any XML

Scanning For Configuration

```
public class HelloWorld {  
    public static void main(String[] args) {  
        AnnotationConfigApplicationContext ac =  
            new AnnotationConfigApplicationContext();  
  
        ac.scan("com.example.config", "com.example.service");  
        ac.refresh();  
  
        Service service = (Service) ac.getBean("greetingService");  
        service.execute();  
    }  
}
```

The scan method takes one or more base packages and scans for annotated application and configuration classes

Composition With @Import and @Autowired

```
@Configuration
public class BeanConfig {

    @Bean
    public GreetingSource source(){
        GreetingSourceImpl s = new GreetingSourceImpl();
        s.setGreeting("Hello World");
        return s;
    }

    @Bean
    public GreetingDestination destination(){
        return new GreetingDestinationImpl();
    }
}
```

Here is the same configuration class we saw earlier,
without the Greeting Service bean


Composition With @Import and @Autowired

```
@Configuration
@Import(BeanConfig.class)
public class ServiceConfig {

    @Autowired
    private BeanConfig beans;

    @Bean
    public Service greetingService(){
        return new GreetingService(beans.source(), beans.destination());
    }
}
```

Configuration classes are also beans and can be the target of dependency injection



Here is the Greeting Service bean configuration in a separate configuration class that imports the configuration class from the previous slide

Composition With @Import and @Autowired

```
public class HelloWorld {  
    public static void main(String[] args) {  
        AnnotationConfigApplicationContext ac =  
            new AnnotationConfigApplicationContext(ServiceConfig.class);  
        Service service = (Service) ac.getBean("greetingService");  
        service.execute();  
    }  
}
```

Since `ServiceConfig` imports `BeanConfig`,
only `ServiceConfig.class` needs to be passed
to the Application Context constructor

Loading Spring in a Web Application

web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <display-name>helloworld</display-name>

  <context-param>
    <param-name>contextClass</param-name>
    <param-value>
      org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
  </context-param>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>com.example.ServiceConfig</param-value>
  </context-param>

  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
  </listener-class>
  </listener>
</web-app>
```

Access Spring Context Staticly:

```
WebApplicationContext ctx =
    WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

Loading Spring in a Servlet 3.0 Container

```
public class HelloWorldWebAppInitializer implements WebApplicationInitializer {  
  
    @Override  
    public void onStartup(ServletContext container) {  
  
        AnnotationConfigWebApplicationContext rootContext =  
            new AnnotationConfigWebApplicationContext();  
  
        rootContext.register(ServiceConfig.class);  
  
        container.addListener(new ContextLoaderListener(rootContext));  
    }  
}
```

As of Spring 3.1, `SpringServletContainerInitializer` will be bootstrapped automatically by any Servlet 3.0 container.

It will find any implementations of `WebApplicationInitializer` on the classpath and call `onStartup`.

Demo 1

- See the JUG_HelloWorld1 project for complete source code from the previous slides

Demo 2

- Life-cycle Mismatch

- Typically, a Singleton bean is injected only when it is first created
- If it is injected with a Prototype bean, it will have the same unique instance of that bean until the application is restarted
- What if the Singleton needs a new instance of the Prototype bean every time it performs an operation?
 - Before Spring 3, solving this problem required adding proxied methods to application classes and non-intuitive XML configuration
- In the JUG_HelloWorld2 project, see HelloWorld.java, ServiceConfig.java, and GreetingDestinationImpl.java

Demo 3

- Use method-chaining with setter methods (3.1)
- Configure a Spring aspect
- Solve the self-referencing object problem
 - Spring uses proxies to implement aspect-oriented programming
 - When a bean is retrieved (or injected) from the application context it might actually be a proxy object
 - If an application object calls methods on itself, it will bypass the proxy and advice will not be applied
 - To avoid this problem, the object needs to retrieve its proxied self from the application context and call methods on the proxy
 - Java-based configuration makes it easier to allow objects to get references from the factory without accessing the application context directly (only works for Singletons)

Demo 3

- In the JUG_HelloWorld3 project, see BobAspect.java, ServiceConfig.java, BeanConfig.java, AOPConfig.java, and GreetingDestinationImpl.java.

Demo 4: Environment and Profiles (3.1)

- The **Environment** is a new abstraction that allows different configurations to be loaded in different deployment environments without modifying Spring configuration files or code
- The **Environment** has a set of active **profiles** that determine which bean configurations load
- Each `<beans>` element, or annotated bean can be assigned to zero or more profiles
 - If none of their assigned profiles are active, they are ignored by Spring
 - If they are not assigned any profiles, they are always loaded

Demo 4

- In the JUG_HelloWorld4 project, see HelloWorld.java and AOPConfig.java.

Setting the Active Profiles (3.1)

- Programmatically (before loading config):
`ac.getEnvironment()
 .setActiveProfiles("foo", "dev");`
- As a system environment variable or JVM system property:
`-Dspring.profiles.active="foo,dev"`
- In web.xml:
`<context-param>
 <param-name>spring.profiles.active</param-name>
 <param-value>foo,dev</param-value>
</context-param>`

Demo 5: Custom Profile Annotations (3.1)

- If profile names are just Strings, how do we prevent typo mayhem?
- Use `@Profile` to meta-annotate custom profile annotations
- In the JUG_HelloWorld5 project, see Dev.java and AOPConfig.java

Demo 6: Environment Properties (3.1)

- The Environment contains a searchable hierarchy of property sources (name-value String pairs)
- The default Environment has this hierarchy:
 - System Properties (highest precedence)
 - System Environment Variables
- You can configure the environment with additional property sources (such as properties files) and assign their order of precedence
- You can inject properties using the `@Value` annotation or use the Environment object

Demo 6: Environment Properties (3.1)

- In the JUG_HelloWorld6 project, see BeanConfig.java and HelloWorld.java

Programmatic Web Context Initialization When Using web.xml (3.1)

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <display-name>helloworld</display-name>

  <context-param>
    <param-name>contextClass</param-name>
    <param-value>
      org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
  </context-param>

  <context-param>
    <param-name>contextInitializerClasses</param-name>
    <param-value>com.example.MyContextInitializer</param-value>
  </context-param>

  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
  </listener-class>
  </listener>
</web-app>
```

Programmatic Web Context Initialization When Using web.xml (3.1)

```
public class MyContextInitializer implements
    ApplicationContextInitializer<AnnotationConfigApplicationContext> {

    @Override
    public void initialize(AnnotationConfigApplicationContext ac) {

        ConfigurableEnvironment env = ac.getEnvironment();

        env.setActiveProfiles("dev");

        ac.scan("com.ocிweb.corespring.helloworld.config");

        env.getPropertySources().addFirst(new MyPropertySource());
    }
}
```

- The ContextLoader calls initialize before it calls refresh() on the context.
- The initialize method can do any programmatic configuration that needs to be done before refresh()

Demo 7: Data

- Embedded Database for development (3.0)
- Hibernate SessionFactoryBuilder APIs (3.1)
- @EnableTransactionManagement (3.1)
- Nested @Configuration classes (3.1)
- See the JUG_DEMO7 project

Demo 8: Cache Abstraction (3.1)

- In the JUG_DEMO8 project, see context.xml, DataConfig.java, and MediaDAO.java
- NOTE!! This example does not run. There is an error parsing context.xml that I haven't figured out yet.

Demo 9: Test Context Support (3.1)

- In the JUG_DEMO9 project, see `ServiceTest.java`

References

- <http://www.springsource.org/documentation>
 - You will find the reference manual and API for Spring 3.1 on this page.
 - You should start by looking at the “What's New in Spring 3” section of the reference manual. The Spring 3.1 subsection will contain links to specific blog postings or API documents that provide further information for each new feature.
- <http://blog.springsource.com/category/spring/31/>