SpringONE

Using Spring 2.0 Custom Namespaces in a Service Oriented Architecture

Brendan Lawlor

DeCare Systems Ireland

SpringONE

Using Spring 2.0 Custom Namespaces in a Service Oriented Architecture

Breandán Ó'Leathlobhair

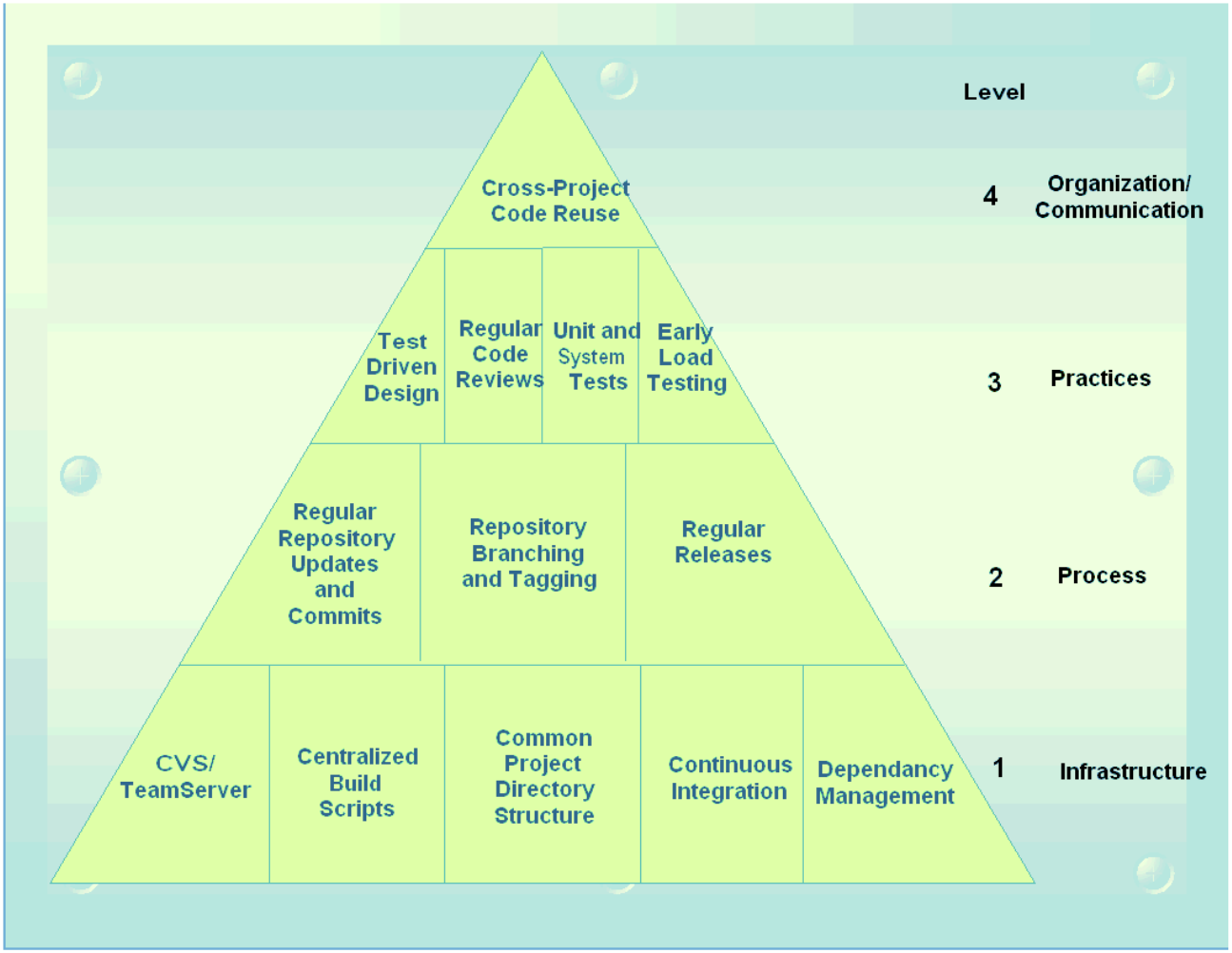
DeCare Systems Ireland

A Little Bit of Technical History

In DSI (DeCare Systems Ireland):

- 2000 – First J2EE Project using JSP
- 2000 – Custom MVC Framework
- 2001 – First EJB 1.1 App with BMP
- 2002 – First EJB 2.0 App with CMP & Apache Struts
- 2003 – Rod Johnson's first book
- 2004 – Agile Development Process
- 2004 – Spring Framework

DSI Process Pyramid



What is IoC?

Inversion of Control: General term, true of most frameworks.

What kind of control is handed over to the Spring Framework?

Dependency Injection: The decision as to what implementations we depend on is left to the framework.

Simple Example: Before

```
import org.apache.commons.dbcp.BasicDataSource;

public class CustomerService {

  BasicDataSource dataSource;
  public CustomerService(){
    dataSource = new BasicDataSource();
  }
}
```

- Coupled to Apache implementation
- Hard to Unit Test

Simple Example: After

```
import javax.sql.DataSource;
public class CustomerService {
  DataSource dataSource;
  public CustomerService (){
  }

  public void setDataSource(DataSource dataSource){
    this.dataSource = dataSource;
  }
}
```

- Decoupled from DataSource implementation
- Mock Object can be substituted for DataSource (easier to unit test)

What we like about Spring (Part I)

- Dependency Injection (IoC):
  - Testability
  - Reusability
  - Dynamic binding (Plug-in capability)
  - Configuration through XML
    - Tool Support
  - Supports Standard Architecture and Design.
- No dependency on Spring Container API!

What we like about Spring (Part II)

- iBATIS
- Hibernate
- Transaction
- Spring's AOP
- Spring & EJB
- Web Services with Apache Axis
- Apache Struts
- Acegi Security
- Drools

Problems that can arise with Spring

- But (there's always a *but*)
  - Concern about proliferation of XML files
  - Difficulty in reusing our own 'Sprung' libraries.

Problem Context

You are designing a Java library.

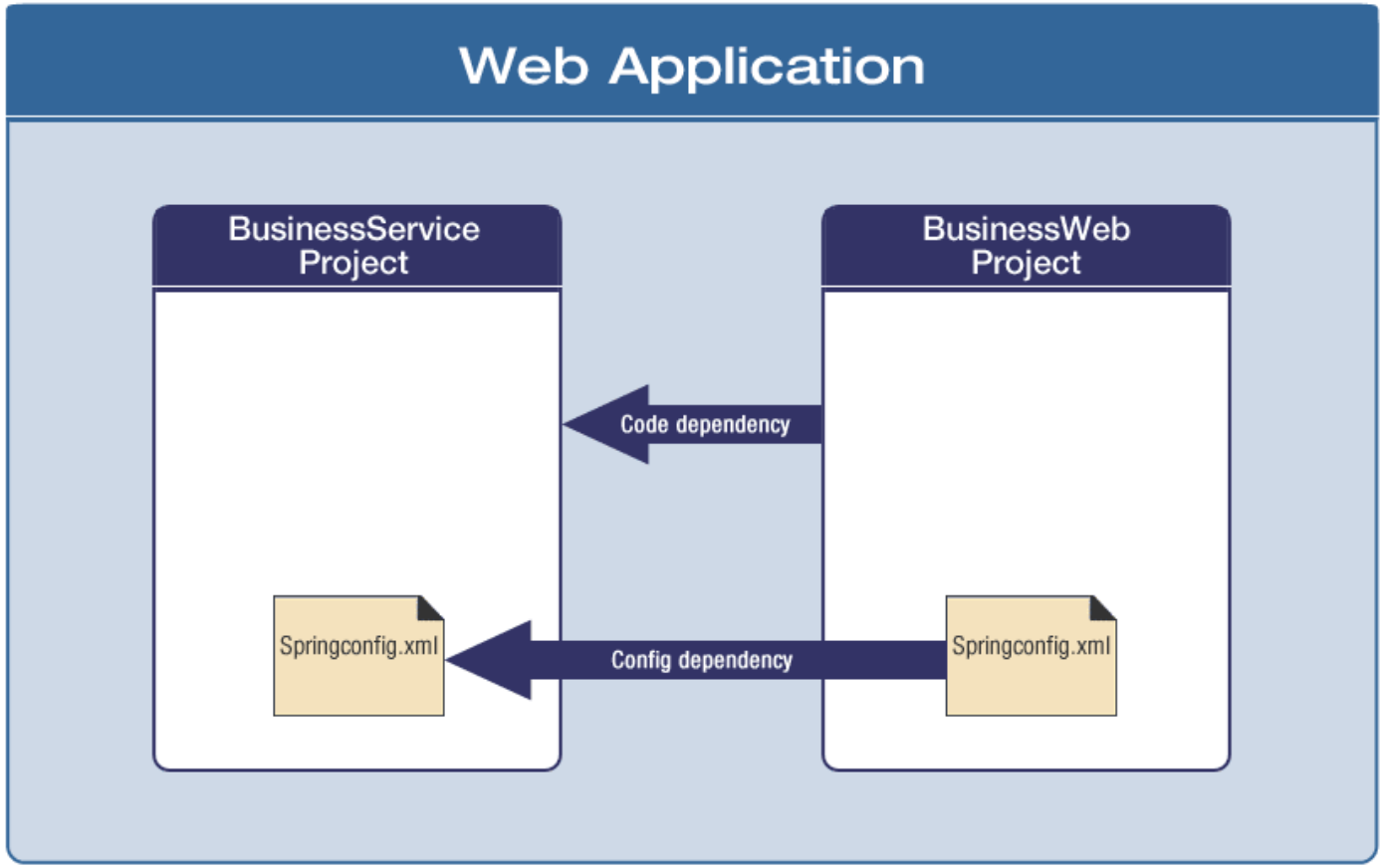
It is implemented using the Spring Framework.

You oblige users of the library to read your Spring configuration files.

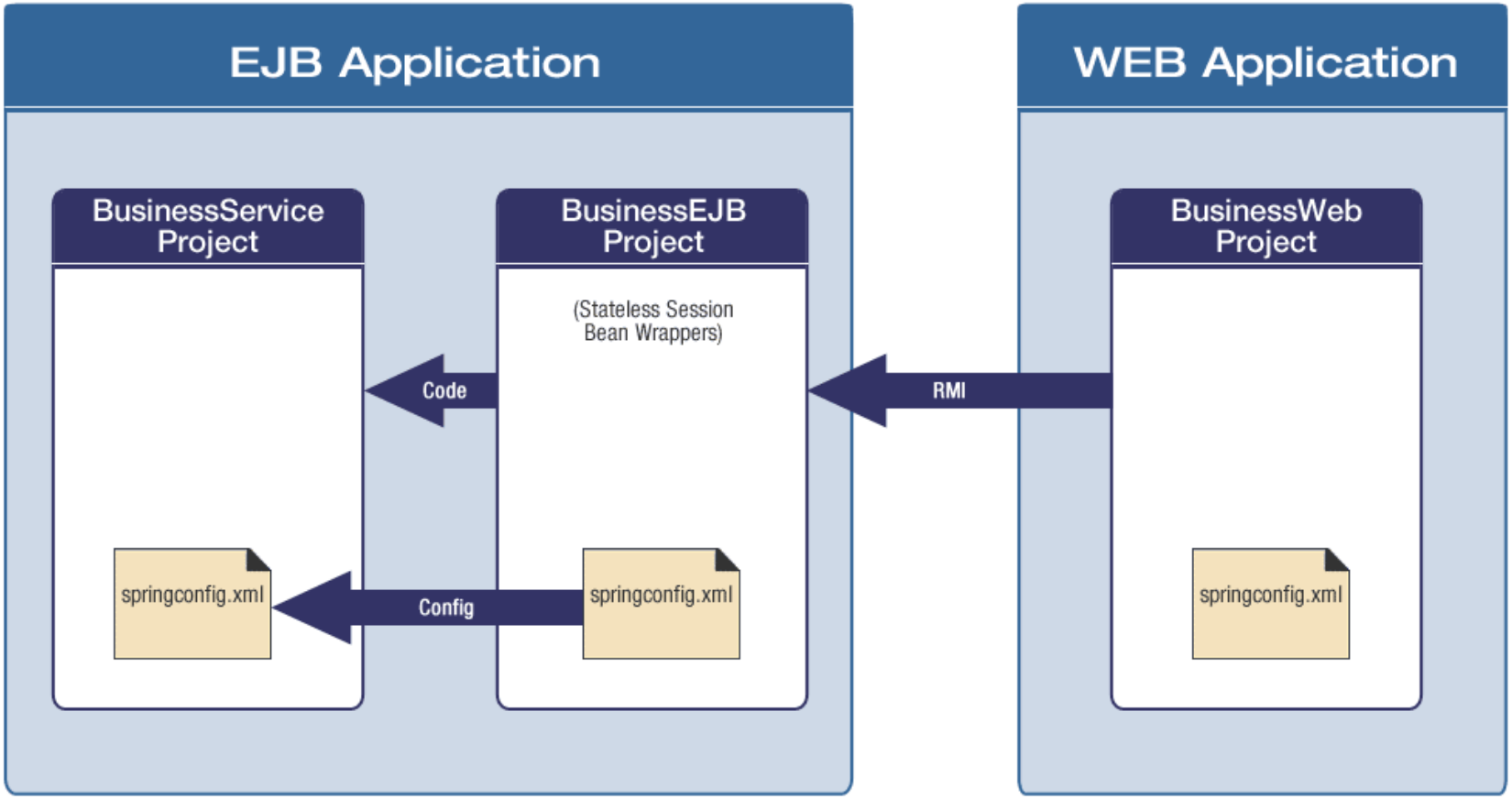
"But I Don't Design Libraries!"

- Very common activity.
- Business logic components as POJO APIs
- Directly compiled into web applications
- Or directly compiled into EJB wrapper projects
- You may be doing this too (or you may want to!)

Example of Business API



Example of Business API



So What's the Problem?

- Spring Framework complicates APIs.
- Jar on classpath isn't enough.
- The layer above needs to know much more than it wants to.
- Application Context is spread across client and API.
- The users of your API are armed, violent and bigger than you.

Example of XML Hell

- Example web.xml segment:

```
...
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:data.xml
                 classpath:services.xml
                 classpath:webContext.xml</param-value>
  </context-param>
  <listener>
    <listener-class>
          org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
...
```

Example of XML Hell

- From API:

```
<bean id="testService" class="…"
 <property name="dataSource">
    <ref bean="testServiceDataSource"/>
  </property>
</bean>
```

- From client:

```
<bean id="testServiceDataSource"
class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="resourceRef">
    <value>false</value>
  </property>
  <property name="jndiName">
    <value>jdbc/ecomds</value>
  </property>
</bean>
```

Solution: First Attempt

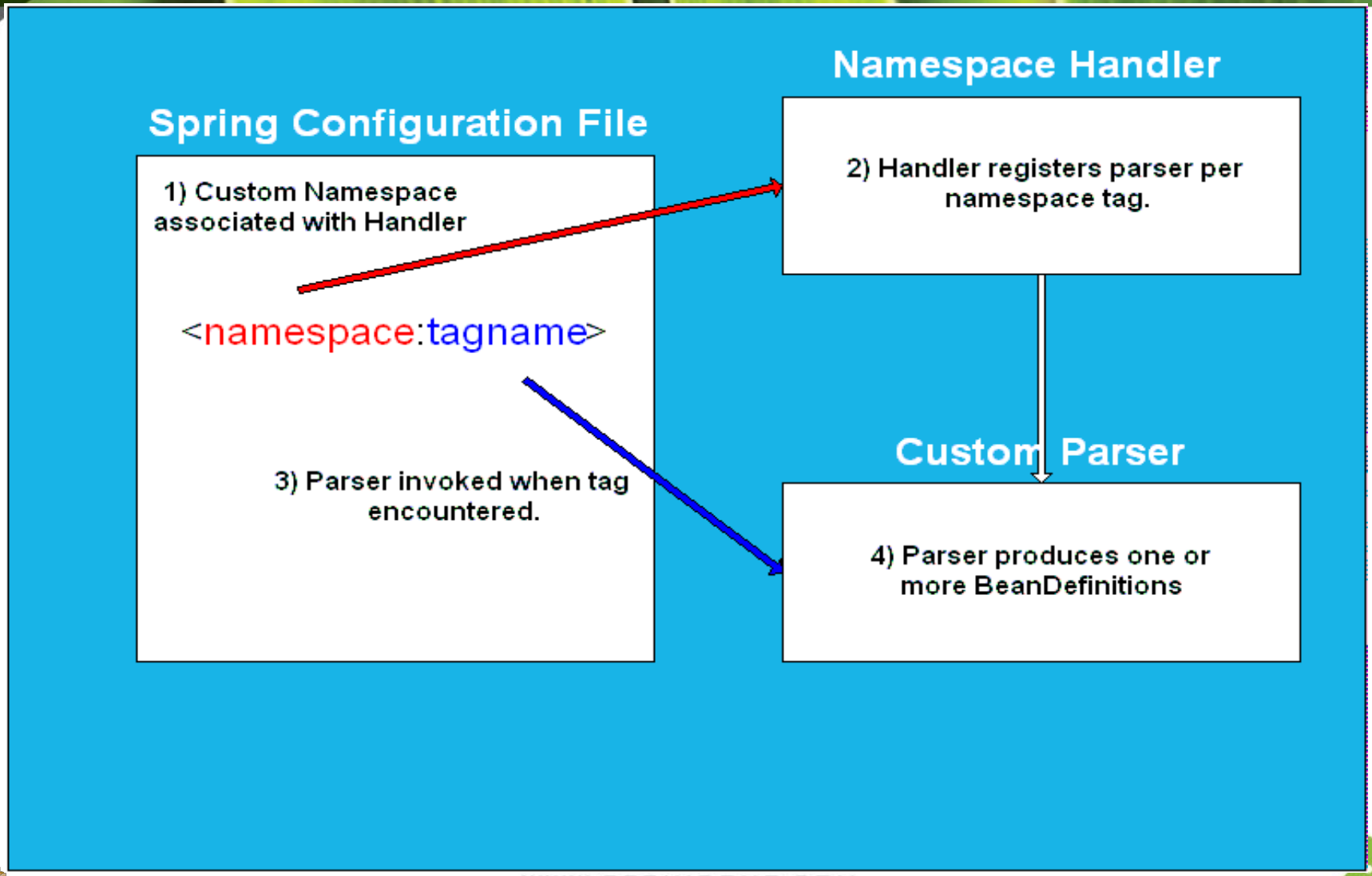
- Standardize, Standardize, Standardize.
  - Use agreed names for configuration files.
  - Use agreed names for public beans.
  - Use agreed names for dangling references.
- Standards like this are boring…
- …but there can be no creativity without form.
- Tooling helps!

Limitations of First Solution

- Still leaks plumbing configuration.
- It medicates the patient…
- …but doesn't deal with the disease:
- Top-level project should specify ApplicationContext.

Spring 2.0 Custom Namespace Example

Spring Custom XML



Solution 2.0

- Example xml:

```
<beans  xmlns="http://....

xmlns:ts="http://www.decaresystems.ie/schema...>

    <ts:myservice id="myService"
       dataSourceRef="customerServiceDataSource">

    <bean id="customerServiceDataSource" class=""…
    />

</beans>
```

Solution: Spring 2.0 Custom XML

- As part of your business API, create an XSD.
- Write a Custom Parser to interpret the XML
- Only expose the public objects…
- …define the remainder silently.
- Only configure the dangling references.
- i.e. Hide the plumbing.

```
public abstract class ServiceNamespaceHandler
extends NamespaceHandlerSupport {

  public TestServiceNamespaceHandler() {
    registerBeanDefinitionParser(
    "testservice"
    new TestServiceDefinitionParser());
  }
}
```

- TestServiceDefinitionParser must code BeanDefinitions for entire API.
- This != Spring

Solution 2.0: Generic Parser

- Write a generic API Definition Parser.
- Extended to…
  - Consume the API configuration files.
  - Register all service beans in client's context.
  - Expose public beans with tags.

API BeanDefinitionParser

```
public class TestServiceNamespaceHandler extends
NamespaceHandlerSupport {

  public TestServiceNamespaceHandler() {

    BeanDefinitionParser parser = new
      ServiceBeanDefinitionParser(
      "/ie/dsi/…/serviceContext.xml");

    registerBeanDefinitionParser("testservice",
                                 parser);
    registerBeanDefinitionParser("otherservice",
                                 parser);
  }
}
```

Solution 2.0

- Example xml:

```
<beans  xmlns="http://....

xmlns:ts="http://www.decaresystems.ie/schema...>

    <ts:testservice id="myService“/>
    <ts:otherservice id="myOtherService“/>


</beans>
```

What About the Dangling References?

- This just defines instances.
- We need to complete/configure them.

```
<beans
xmlns="http://....          xmlns:ts=http://
www.decaresystems.ie/schema/testservice>

    <ts:testservice id="myService"
        dataSourceRef="testServiceDataSource"/>

    <bean id="testServiceDataSource"
          class="…/>

</beans>
```

Testing Against Reality

- Can't assume references are public bean properties.
- Need to map:
  - public tag/attribute pairs to
  - corresponding bean/property pairs.

Custom Properties

```
<bean id="servicePropertyRedirects"
    class="ie.dsi.springschema.PropertyRedirecter">
  <property name="redirects">
    <list>
      <bean class="ie.dsi.springschema.PropertyRedirect">
        <property name="sourceElementName" value="email"/>
        <property name="sourceAttributeName"
                  value="smtpHostname"/>
        <property name="targetBeanName" value="mailSender"/>
        <property name="targetBeanProperty" value="host"/>
      </bean>
    </list>
  </property>
</bean>
```

Now we have everything we need!

- This redirection could be coded in extended ServiceBeanParser.
- In fact you can, but we prefer the Spring way.
- Next step would be to simplify it with…
- …custom XML!

Issues and Future Directions

- Assumes ‘Singleton’
- Still theoretical possibility of name collision.
- Limitations of Namespaces in Spring
- Candidate for Tooling (e.g. Eclipse plugin)

Resources

- www.springone.com
- blog.decaresystems.ie
- erik.jteam.nl/?p=23
- Spring JIRA MOD-164


Q&A