

# Using Spring and AspectJ in a CaptainCasa project

- 1 What about this AOP stuff?
- 2 Requirements
- 3 Adding Spring to your project
- 4 AOP with CaptainCasa

## 1 What about this AOP stuff?

Currently there is a big hype about AOP. While several people are already talking about AOP being the next programming methodology after object oriented programming, others keep saying it's just some fancy stuff without any impact on developers...

Well, actually we should not care about others opinions regarding AOP and simply take a closer look to AOP, some might benefit from AOP in their project, others might not.

The basic idea of AOP is, that there are „tasks“ (so called crosscutting concerns) which are repeated periodically within an application. Logging is a good example, you might have in any of your „onDoSomething...()“ methods a log statement:

```
private static Logger log = Logger.getLogger>HelloBean.class);

public void onLogin()
{
    log.info("Logging in...");
    // ...
}

public void onGetItems()
{
    log.info("Fetching items...");
    // ...
}
```

The logging is implemented through the whole application (of course not only in your managed beans, usually it's tied down to your services...), so in your code there is a mixture of Logging and (business) logic - not a big thing, but could be done better, and that's exactly where AOP comes in. AOP enables you to isolated thes periodical operations and of course logging is only one example for isolating redundant operations. The same could be done to Exception Handling, validation etc...

These redundant operations are tied together in aspect classes. An aspect for handle logging for a method *onHello()* might look like this:

```
@Aspect
public class MyAspect
{
    // prepare log4j logging
    private static Logger log = Logger.getLogger(MyAspect.class);

    @Before("execution(* managedbeans.HelloBean.onHello(..)")
    public void logSayHelloStart()
    {
        log.info("Starting say hello" + new Date().getTime());
    }

    @After("execution(* managedbeans.HelloBean.onHello(..)")
    public void logSayHelloEnd()
    {
        log.info("Finishing say hello" + new Date().getTime());
    }
}
```

```
}  
}
```

We will take a closer look to the annotations later on, but in order to understand AOP you should simply imagine AOP being kind of „Listener“. You can configure this Listener like „...listen and invoke when methodABC is entered...“ or „...listen and invoke on any method call in package/class XYZ...“.

Thats all the magic of AOP, sorry to those who expected „more“ and congrats to those who finally understood first time what AOP is about! ;-)

## 2 Requirements

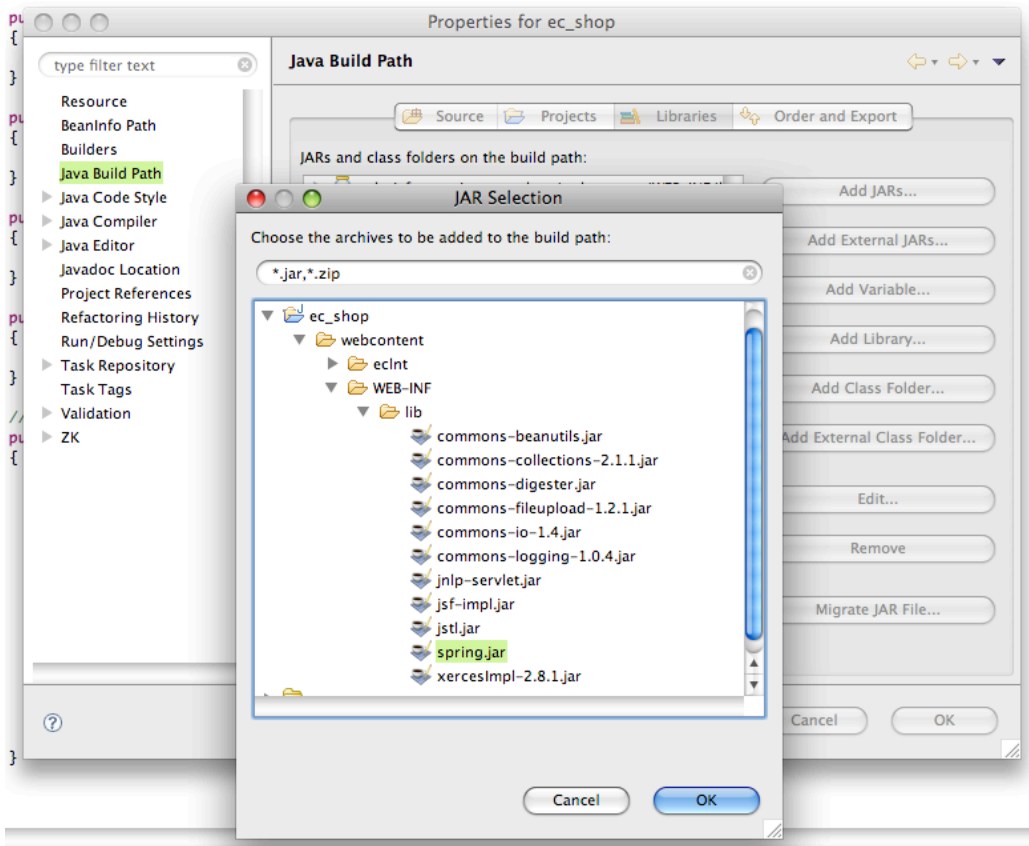
There are several frameworks available to implement AOP. One of the common ones is AspectJ, the nice thing about AspectJ is: it can directly integrate with Spring. Using a „naked“ AOP framework usually results in adapting your application server, that's ugly and time wasting, so we decided for the smart Spring solution...

Download the following packages to your system (best practice: create a new directory aop somewhere and store all AOP related stuff to this folder...):

- AspectJ - <http://www.eclipse.org/aspectj>(careful! extract the downloaded .jar since the jar is a zip - it contains docs, distribution code and the required jars, throwing in the whole jar into your project later on will not work...)
- AspectJ Eclipse plugin - <http://www.eclipse.org/ajdt/> (simply install and restart Eclipse)
- Spring - <http://www.springsource.org/>

## 3 Adding Spring to your project

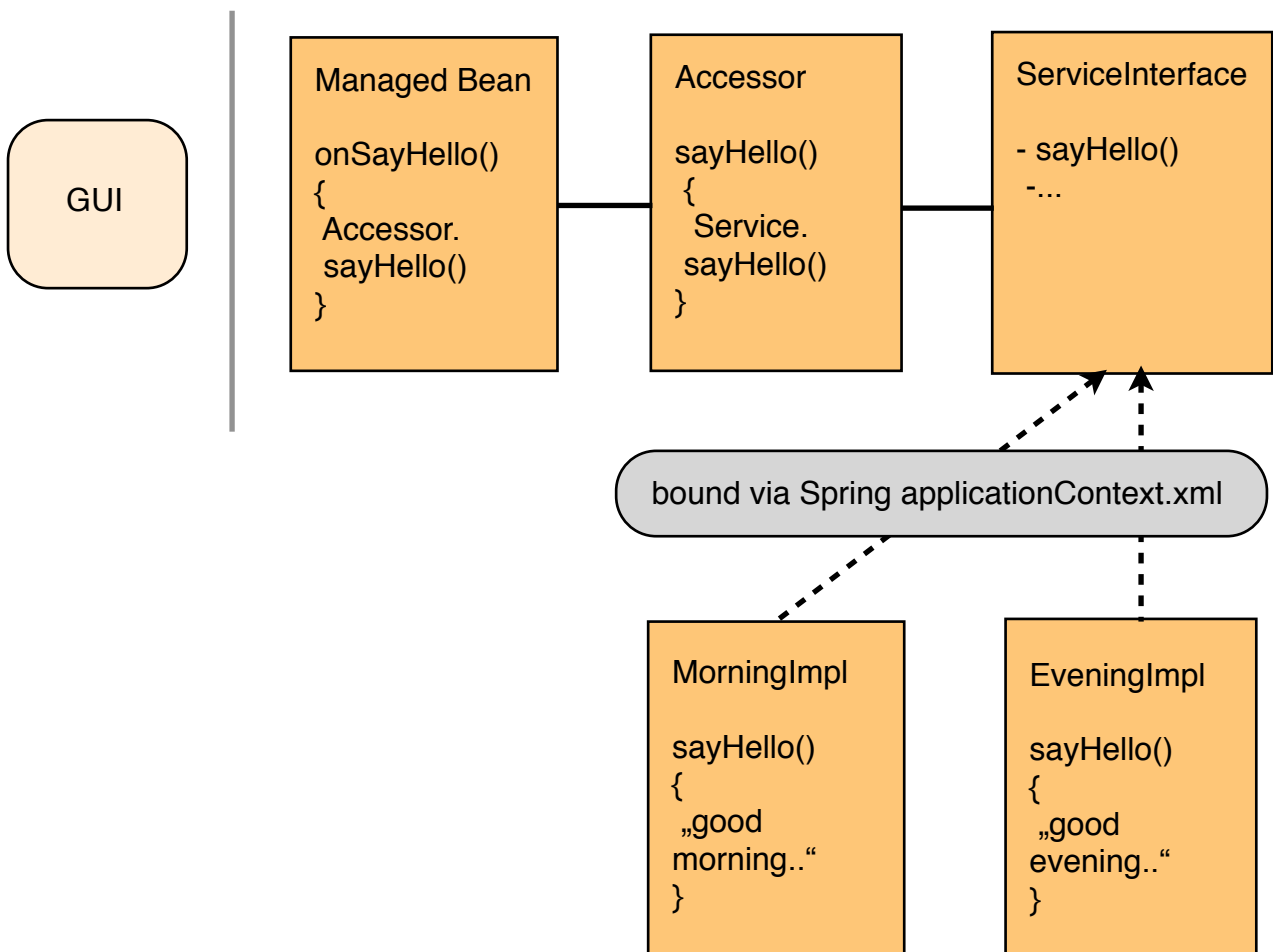
Before we actually start with AOP we have to ensure Spring is integrated to our project. We assume you already have created a CaptainCasa project and imported the project to Eclipse. Within your Eclipse environment drag&drop the file spring.jar from your Explorer/ Finder to the folder *webcontent/WEB-INF/lib* and finally add spring.jar to your build path (right mouse button on project - Properties - Java Build Path). Click the Libraries tab and press the *add jar* button, navigate down to the WEB-INF/lib folder of your project and select spring.jar (Screenie 1).



**Screenie 1: add spring.jar to build path of your project.**

Now you should be able to implement Spring related stuff in your project, and that is exactly what we will be doing next. We will create a service binding from the CaptainCasa managed bean to a Java service. This could be done by „hard-wiring“ the service in our bean - or Spring can be used in order to configure a loose XML binding between bean and service...guess which way is the preferred one! ;-)

Please refer to the scribble below, it will give you a quick overview about the „backend“ - it's not UML-conform (currently dont have an UML tool with me...) but you should be able to understand it anyway since this is essential Java usage...



**Scribble: High level „architecture“...**

Usually we would have a „hard wired binding“ in the Accessor, defining the service implementation we actually like to use. Thanx to Spring we do not have to do so, the binding is configured via applicationContext.xml. Please refer to the codings below in order to implement the Spring stack...:

**The Service Interface (HelloService.java)**

```

package services;

public interface HelloService
{
    public String sayHello(String username);
}

```

## The Service Implementations (HelloEveningServiceImpl.java/ HelloMorningServiceImpl.java)

```
package services;

public class HelloEveningServiceImpl implements HelloService
{
    public String sayHello(String username)
    {
        return "Good evening, " +username;
    }
}
```

```
package services;

public class HelloMorningServiceImpl implements HelloService
{
    public String sayHello(String username)
    {
        return "Good morning, " +username;
    }
}
```

## The Accessor (HelloServiceAccessor.java)

```
package serviceAccessors;

import services>HelloService;

public class HelloServiceAccessor implements HelloService
{
    HelloService helloService;
    public void setHelloService(HelloService service) { helloService = service; }

    public String sayHello(String username)
    {
        return helloService.sayHello(username);
    }
}
```

## The Managed Bean (HelloBean.java)

```
public class HelloBean implements Serializable
{
    // service accessor
    private HelloServiceAccessor serviceAccessor;

    // fields
    private String m_name;
    private String m_output;

    public String getOutput()
    {
        return m_output;
    }

    public void setOutput(String output)
    {
        m_output = output;
    }
}
```

```

public String getName()
{
    return m_name;
}

public void setName(String name)
{
    m_name = name;
}

// actions
public void onHello(ActionEvent ae)
{
    // set accessor
    if (serviceAccessor == null)
    {
        ApplicationContext context = new ClassPathXmlApplicationContext("META-INF/
applicationContext.xml");
        serviceAccessor =
(HelloServiceAccessor)context.getBean("HelloServiceAccessor");
    }

    if (m_name != null)
    {
        m_output = serviceAccessor.sayHello(m_name);
    }
    else
    {
        m_output = "Enter name!";
    }
}
}
}

```

Ok, the services as well as the service implementations are pretty straight forward. The Managed Bean requires some explanations: within the *onHello*-Action a *serviceAccessor* is instantiated, the instance is not create via „*accessor = new ServiceAccessor()*...“, Spring is used for the instantiation. Of course Spring does not only instantiate the accessor, Spring also populates the service implementation within the accessor. Therefore the file *META-INF/applicationContext.xml* is required in your source folder, and the Spring configuration should look like this:

## applicationContext.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd>

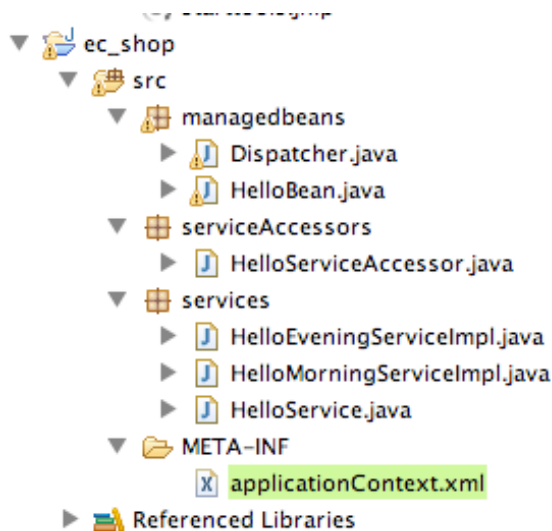
<bean id="HelloMorningServiceImpl" class="services.HelloMorningServiceImpl" />
<bean id="HelloEveningServiceImpl" class="services.HelloEveningServiceImpl" />

<bean id="HelloServiceAccessor" class="serviceAccessors.HelloServiceAccessor">
    <property name="helloService" ref="HelloMorningServiceImpl" />
</bean>

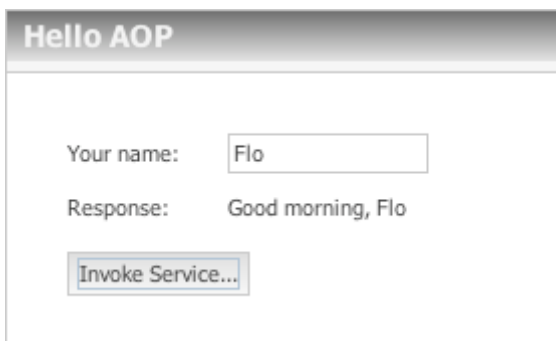
</beans>

```

You got it? The two service implementations are declared as beans as well as the ServiceAccessor, the Accessor has a property and this property simply defines which implementation should be used. The ServiceAccessor contains SETTER/GETTER which will be used by Spring in order to inject the service implementation..



The server part of your project now should look like this:



Last but not least, create a simple UI and link HelloBean.onSayHello to a button or something similar, define some input/output properties in order to display the result:

### Corresponding JSP code:

```
<f:view>
<h:form>
<f:subview id="helloSpringg_sv">
<t:rowtitlebar id="g_1" text="Hello AOP" />
<t:rowheader id="g_2" />
<t:rowbodypane id="g_3" >
<t:rowdistance id="g_4" height="10" />
```

```

<t:row id="g_5" >
<t:coldistance id="g_6" width="10" />
<t:label id="g_7" text="Your name:" width="80" />
<t:field id="g_8" text="#{HelloBean.name}" width="100" />
</t:row>
<t:rowdistance id="g_9" height="10" />
<t:row id="g_10" >
<t:coldistance id="g_11" width="10" />
<t:label id="g_12" text="Response:" width="80" />
<t:label id="g_13" text="#{HelloBean.output}" width="100" />
</t:row>
<t:rowdistance id="g_14" height="15" />
<t:row id="g_15" >
<t:coldistance id="g_16" width="10" />
<t:button id="g_17" actionListener="#{HelloBean.onHello}" text="Invoke
Service..." />
</t:row>
</t:rowbodypane>
<t:rowstatusbar id="g_18" />
<t:pageaddons id="g_pa"/>
</f:subview>
</h:form>
</f:view>

```

Now you should be able to access your service implementations through Spring; simply exchange *Morning Service* against *Evening Service*, greetings should change immediately.

Cool stuff! Finally the last step is integratin of AOP.





## 4 AOP with CaptainCasa

Let's implement the following scenario based on AOP: any on...() method within the managed bean should be traced by logging the method entrance time as well as the method exit time; delta of course is time consumed during the method call...

```

Starting execution(void managedbeans.HelloBean.onHello(ActionEvent)):
1242812079671
Finishing execution(void managedbeans.HelloBean.onHello(ActionEvent)):
1242812080410

```

 aspectjrt.jar	2. April 2009, 09:04
 aspectjtools.jar	2. April 2009, 09:04
 aspectjweaver.jar	2. April 2009, 09:04
 org.aspectj.matcher.jar	2. April 2009, 09:04

First thing you need to do for AOP usage is

adding the AspectJ jar files to your project's lib folder, dont forget to link the four jars to the build path of your project (Project - Properties etc.)

Next thing is to add AspectJ support to your project. Make sure you have the AspectJ Eclipse plugin installed, if so, select right mouse button on your project - *AspectJ Tools - Convert to AspectJ project*.

No you will be able to use Aspect annotations in your project, create a new class (=Aspect) with the contents displayed below:

```

package aop;

import java.util.Date;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class MyAspect
{
    // with log4j (the "professional way...")
    // private static Logger log = Logger.getLogger(MyAspect.class);

    @Before("execution(* managedbeans.HelloBean.on*(..))")
    public void logStart(JoinPoint thisJoinPoint)
    {
        System.out.println("Starting " +thisJoinPoint.toString() +": " + new
Date().getTime());
        //log.info("Starting " +thisJoinPoint.toString() +": " + new Date().getTime());
    }

    @After("execution(* managedbeans.HelloBean.on*(..))")
    public void logEnd(JoinPoint thisJoinPoint)
    {
        System.out.println("Finishing " +thisJoinPoint.toString() +": " + new Date().getTime());
    }
}

```

```

    // log.info("Finishing " +thisJoinPoint.toString() +": " + new Date().getTime());
    }
}

```

In the class there are two s called „JoinPoints“ defined (remember the listener comparison). Any time a method starting with „on“ is invoked, the first method *logStart()* is invoked BEFORE the method is executed (@Before annotation), the second method *logEnd()* is invoked AFTER execution (@After annotation) of a method starting with „on“.

There are tons of possibilities for defining JoinPoint filters (the current filter is a filtering all methods starting with on...), you can easily create a filter for a single method or a group of methods ending with „MyDesiredEnd()“:

```
@After("execution(* managedbeans.HelloBean.*MyDesiredEnd(..)")")
```

A good page with various Filter examples and corresponding explanations can be found here: <http://static.springframework.org/spring/docs/2.5.x/reference/aop.html>

Please notice, it's not the goal of this document to explain AOP in depth, primary goal is to give you a quick ramp up in order to use AOP with CaptainCasa initially and extend your AOP skills on your own - so simply refer to the link above and you will quickly become an AOP-hero!

Finally we have to register the aspect within the Spring applicationContext.xml, so add the following lines to your applicationContext.xml:

```

<aop:aspectj-autoproxy/>

<bean id="myAspect" class="aop.MyAspect">
    <!-- configure properties of aspect here as normal -->
</bean>

```

Restart the server and test the application - the application now should give you exact entrance and exit time for the *onSayHello()* method!

```

</t:row>
</t:rowbodypane>
<t:rowstatusbar id="g_18" />
</jsp>
Starting execution(void managedbeans.HelloBean.onHello(ActionEvent)): 1242812079671
Finishing execution(void managedbeans.HelloBean.onHello(ActionEvent)): 1242812080410

```

So you got the idea behind AOP? Then simply start learning AOP by using it! :-)