COLDSPRING

BETTER LIVING THROUGH CONFIGURATION

Mark Drew
http://www.markdrew.co.uk

Welcome to the first meeting of

SPAGHETTI CODERS ANONYMOUS (SCA)

What this presentation IS about

- Understanding ColdSpring
- Getting to grips with Dependency Injection
- How to write (more) reusable code
- How to write (more) configurable code
- Laughing at typos made during the presentation
- Helping me spot typos so that I don't spend 10 minutes debugging something

What this is not about

- A framework battle
- A headache inducer
- How to cope with other developer's badly written code (search wikipedia for **Spanish Inquisition Torture Methods** for more details)

You never expect them do you?

Who is Mark Drew?

- CFEclipse's Lead Developer
- Adobe Community Expert
- Product Manager at Design UK http://www.designuk.com developing Lava Suite. A CMS/CRM/eComm application
- UK ColdFusion User Group's Co-Manager
- A ColdFusion Developer since '97 and Web developer since '94
- Reactor ORM Project Manager
- A reformed Spaghetti coder

CFEclipse

What *IS* ColdSpring

- An Inversion of Control Framework
- or.. A Dependency Injection Framework
- Based on the Java Spring Framework (not a port, doesn't include the MVC stuff)
- A component factory (named a bean factory... Tasty!)

Not to be used in enclosed spaces

Inversion of Control
and Dependency Injection

- It INJECTS dependencies into your components for you
- Why? Code should be re-usable, although good, it means you have to wire more together (aka: coupling)
- Less coupling is better practice, your components shouldn't need to know how to create their own dependencies
- You just ask the BeanFactory for your objects and they come all ready for you

Ready for the show!

Real World Example

- What do I need to get to work?
- ... an Assistant!
- ...That is, a bean factory, it prepares things "to do work" so I don't have to.
- My iPod ... but they need headphones
- My cigarettes... but they need a lighter
- And my jacket...

My Prep List

- IPod needs headphones
- Cigarettes need a lighter (which needs fuel, but developers and flammable liquids don't go together)
- My Jacket needs the IPod and Cigarettes to be in it

Real Life Summary

- I have "inverted" who is controlling the supply of my objects
- I have let this "inverted" controller "inject" the dependencies into my objects

ColdFusion Example (ShoppingBasket.cfc)

```
<cffunction name="init" returnType="Any" output="false">
    <cfargument name="MaxItems" type="numeric">

    <!---  We need the TaxCalculator to do figure out the tax, so lets create it --->
    <cfset this.TaxCalculator = CreateObject("component", TaxCalculator).init()>
    <!--- Set the local variable --->
    <cfset this.MaxItems = arguments.MaxItems>
    <cfreturn this/>
</cffunction>
```

What's wrong with that?!

What happens if ...

- What happens if the TaxCalculator changes?
- Lets add a Tax Rate to the Tax Calculator
- (Dont even get me started on the tax increases on cigarettes and alcohol!)

Brain the size of a Planet and I have to deal with taxes??

Example: ShoppingBasket with Tax Calculator

```
<cffunction name="init" returnType="Any" output="false">
    <cfargument name="MaxItems" type="numeric">
    <cfargument name="TaxRate" type="numeric">
    <!---  We need the TaxCalculator to do figure out the tax, so lets create it --->
    <cfset this.TaxCalculator = CreateObject("component", TaxCalculator)
            .init(arguments.TaxRate)>
    <!--- Set the local variable --->
    <cfset this.MaxItems = arguments.MaxItems>
    <cfreturn this/>
</cffunction>
```

Don't over do it or anything...

What is wrong with this?

- The TaxCalculator's constructor changed.
- The basket is now creating and dealing with the TaxCalculator and needs a TaxRate
- It is not really in the remit of the basket
- If we change it again, we have to go and change our code
- Lets clean it up a bit

Remove the constructor...

```
<cffunction name="init" returnType="Any" output="false">
    <cfargument name="MaxItems" type="numeric">
    <cfargument name="TaxCalculator" type="TaxCalculator">
    <!--- We are expecting a configured Tax Calculator --->
    <cfset this.TaxCalculator = arguments.TaxCalculator>
    <!--- Set the local variable --->
    <cfset this.MaxItems = arguments.MaxItems>
    <cfreturn this/>
</cffunction>
```

What did we do there?

- Removed the coupling between the Basket and the TaxCalculator
- The dependency is still there
- The Basket now just expects a ready TaxCalculator
- It doesn't care how the TaxCalculator is "built"

Separation of Concerns

- Components don't have to do things that are not related to their function
- Components aren't tied to other components' implementations
- Components are easier to configure without changes to the code

But...

- We just moved the responsibility to our calling code
- We changed our code from this:

```
<cfset Basket = CreateObject("component", "Basket").init(10)>
```

- To this:

```
<cfset TaxCalculator = CreateObject("component", "TaxCalculator").init(1.175)>
<cfset Basket = CreateObject("component", "Basket").init(10, TaxCalculator)>
```

What if...

- There was a way to manage this "wiring" code?
- Ohh... I know... ColdSpring!
- Our calling code would look like:

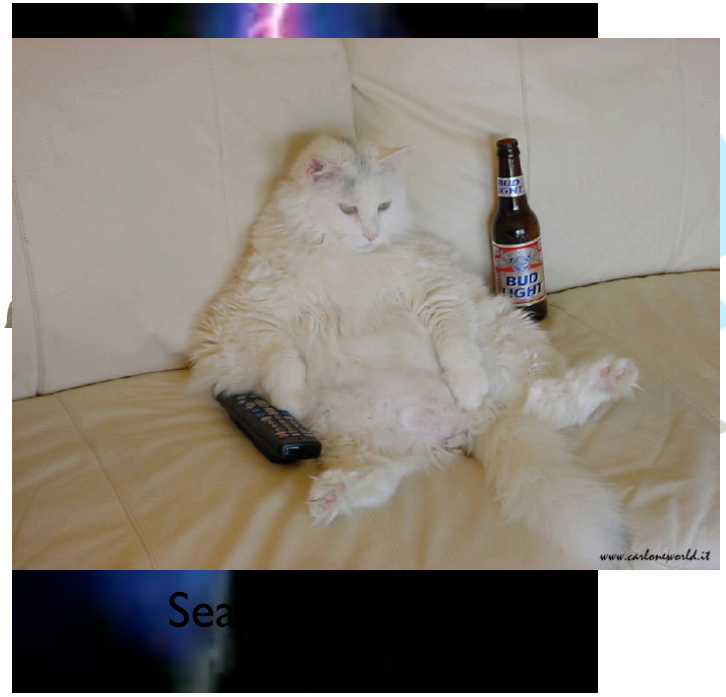
```
<cfset Basket = bf.getBean("Basket")>
```

ColdSpring

- ColdSpring would create the dependencies and wire them together
- We don't need to change the code in all our components that use TaxCalculator
- There is no need to do special coding (no special attributes)
- And we can see our dependencies in one place!
- We have been saved by Dependency Injection!

Quick Terminology

- Bean = Component
- BeanFactory = ColdSpring, the holder of the beans
- Singleton = There can be only one
- LazyLoad = Load it when its called not on startup



The BeanFactory

- Holds all your components
- Instantiates, configures and resolves dependencies for your components (beans)
  - e.g. A ShoppingBasket needs a TaxCalculator that needs a Config

Implementing ColdSpring

- Setup with a simple XML File
- Based on the Java-beans spec (hence we call components <beans>)
- Defines the “beans” (i.e. Components) and what they need to be initialised.

Implementing ColdSpring

```
<beans>

    <bean id="ShoppingBasket"
          class="presentation.cs.ShoppingBasket"
    />

</beans>
```

- Configure “presentation.cs.ShoppingBasket” and when I ask for “ShoppingBasket” return an initialized instance of the component

Impementing ColdSpring

- As a constructor (`...init(MaxItems)`)

```
<beans>

    <bean id="ShoppingBasket"
          class="presentation.cs.ShoppingBasket">

        <constructor-arg name="MaxItems">
            <value>10</value>
        </constructor-arg>

    </bean>

</beans>
```

Impementing ColdSpring

- As a property (`...setMaxItems(MaxItems)`)

```
<beans>

    <bean id="ShoppingBasket"
          class="presentation.cs.ShoppingBasket">

        <property name="MaxItems">
            <value>10</value>
        </property>

    </bean>

</beans>
```

Implementing our Example

```
<beans>
    <!-- define the Tax Calcula
    <bean id="TaxCalculator" class="presentation.cs.TaxCalculator">
        <constructor-arg name="TaxRate"
            <value>1.175</value
        </constructor-arg>
    </bean>

    <bean id="ShoppingBasket"
          class="presentation.cs.ShoppingBa
          <constructor-arg name="MaxItems">
          <constructor-arg name="TaxCalculat
            <ref bean="TaxCalculator" />
          </constructor-arg>
    </bean>

</beans>
```

Set up the TaxCalculator

Set up the ShoppingBasket

Pass in the configured TaxCalculator

Benefits:

- No code was needed in the ShoppingBasket to setup the TaxCalculator
- Your code is now more reusable, testable and maintainable
- Go you!

Configuring YOUR beans

- Not all components are as nicely written as the Shopping Basket
- Different components require different configuration options
- How do we pass arrays and structures to our existing components?

Data Types

- Components can be configured with:
  - Strings (value)
  - Arrays (list)
  - Structures (map)
  - Other components (ref bean)

Strings:

- You have seen this before:

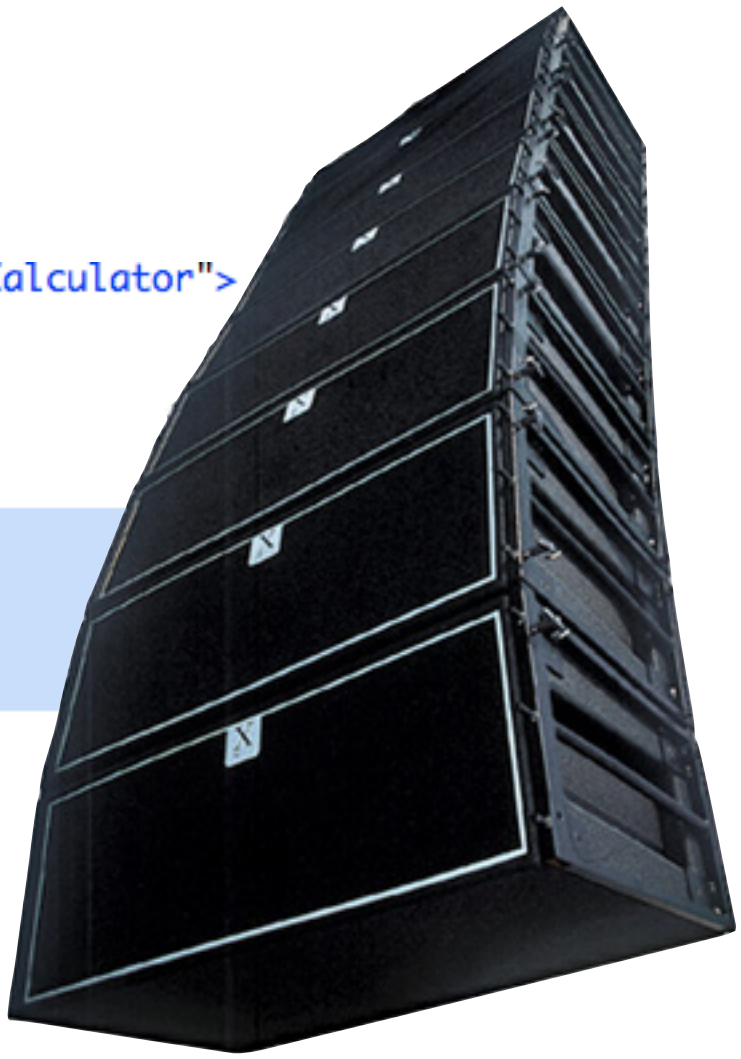
```
<bean id="TaxCalculator" class="presentation.cs.TaxCalculator">
    <constructor-arg name="TaxRate">
        <value>2</value>
    </constructor-arg>
</bean>
```

Arrays:

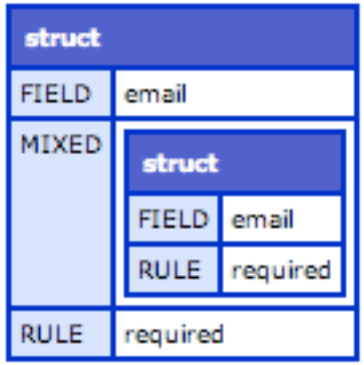
- Also known as a “list”

```
<bean id="TaxCalculator" class="presentation.cs.TaxCalculator">
    <constructor-arg name="TaxRate">
        <value>2</value>
    </constructor-arg>
    <constructor-arg name="DeliveryCountries">
        <list>
            <value>UK</value>
            <value>US</value>
            <value>EUROPE</value>
        </list>

    </constructor-arg>
```



Structures:



- Also known as a “map”

```
<bean id="BeanExample" class="presentation.cs.Bean">
    <constructor-arg name="maps">
            <map>
                <entry key="field"><value>email</value></entry>
                <entry key="rule"><value>required</value></entry>
                <entry key="Mixed">
                    <map>
                        <entry key="field"><value>email</value></entry>
                        <entry key="rule"><value>required</value></entry>
                    </map>

                </entry>
            </map>
    </constructor-arg>
</bean>
```

Components:

- You have seen this before in the Shopping Basket
- Known as a “ref bean” i.e.: A reference to a bean

```
<bean id="BeanExample" class="presentation.cs.Bean">

    <constructor-arg name="Reference">
        <ref bean="Reference" />
    </constructor-arg>

</bean>
```

Auto-Magic Wiring

- You don't need to explicitly pass beans to your components
- Using accessors (getters and setters) ColdSpring will pass in objects if required
- set<BeanName> in your component is all that is required.
- and autowire="byName" or autowire="byType"

Auto-Magic Wiring...

- setTaxCalculator called and it will pass the TaxCalculator Bean
- or
- init(TaxCalculator) injects by type of your init arguments

Got it?

- Questions?

(The really difficult) Installation

- Download it from
  - http://www.coldspringframework.org/
  - Copy to your webroot
  - or
  - Create a mapping called “coldspring”

Implementation

```
<!--- Create a bean factory --->
<cfset myBeanFactory = CreateObject("component",
            "coldspring.beans.DefaultXmlBeanFactory").init()>
```

Configuration

- The DefaultXmlBeanFactory can only read an XML formatted configuration
- Currently no implementation for the runtime addition of bean settings
- You can't change the configuration at runtime... really... its dangerous
- BUT: XML can be constructed and passed in

Passing in XML

- Configuration can be passed in as:
  - A fully qualified path
  - Raw XML as a string
  - An XML Object

Passing a File Path

```
<!--- As a Path --->
<cfset myBeanFactory.loadBeansFromXmlFile(expandPath("Coldspring.xml")
                    , true)>
```

true = Lazy Initalisation, e.g. dont load the beans unless they are called

Passing a Raw XML String

```
<!--- As a raw xml string --->
<cfsavecontent variable="beanConfig">
<beans>
    <bean id="aBean" class="app.model.SomeComponent" />
</beans>
</cfsavecontent>
<cfset myBeanFactory.loadBeansFromXmlRaw(beanConfig
                    , true)>
```

Passing an XML Object

```
<!--- As an XML object --->
<cffile action ="read" file="#expandPath("Coldspring.xml")#"
        variable="xmlContent">

        <cfset beanXML = XMLParse(xmlContent)>
<cfset myBeanFactory.loadBeansFromXmlObj(beanXML
                    , true)>
```

Getting your objects

```
<!--- Create a bean factory --->
<cfset myBeanFactory = CreateObject("component",
            "coldspring.beans.DefaultXmlBeanFactory").init()>

<!--- As a Path --->
<cfset myBeanFactory.loadBeansFromXmlFile(expandPath("Coldspring.xml")
                    , true)>

<!--- Get the object --->
<cfset myBasket = myBeanFactory.getBean("ShoppingBasket") />
```

Demo

- Time to prove it!

AOP Anyone?

- Aspect Oriented Programming
- Some functionality applies across many different parts of your code (logging, versioning, security, etc...)
- Code can become entangled, and repetitive functions appear in your components
- Different ‘aspects’ of your code can be wired together to execute without affecting each other

AOP Continued...

- Call your beans as normal
- You can intercept method calls and "introduce" new functionality with Advisors
- No change needs to be done to your bean
- Advisors have functions to run before, during and after selected method calls in your component

Demo

- Lots of new language coming up! (but its OK! Calm Down!)

"I sed.... Calm down.. Calm down!"

Web-Services (Remote Proxies)

- Generate Web-Services from your Beans
  - Creates a Remote Façade to your Beans
  - Select methods to expose
  - Generates a cfc with remote functions (yes! An actual CFC file! OMG!)

Demo

- Lets expose documents to the world using a Remote Proxy!

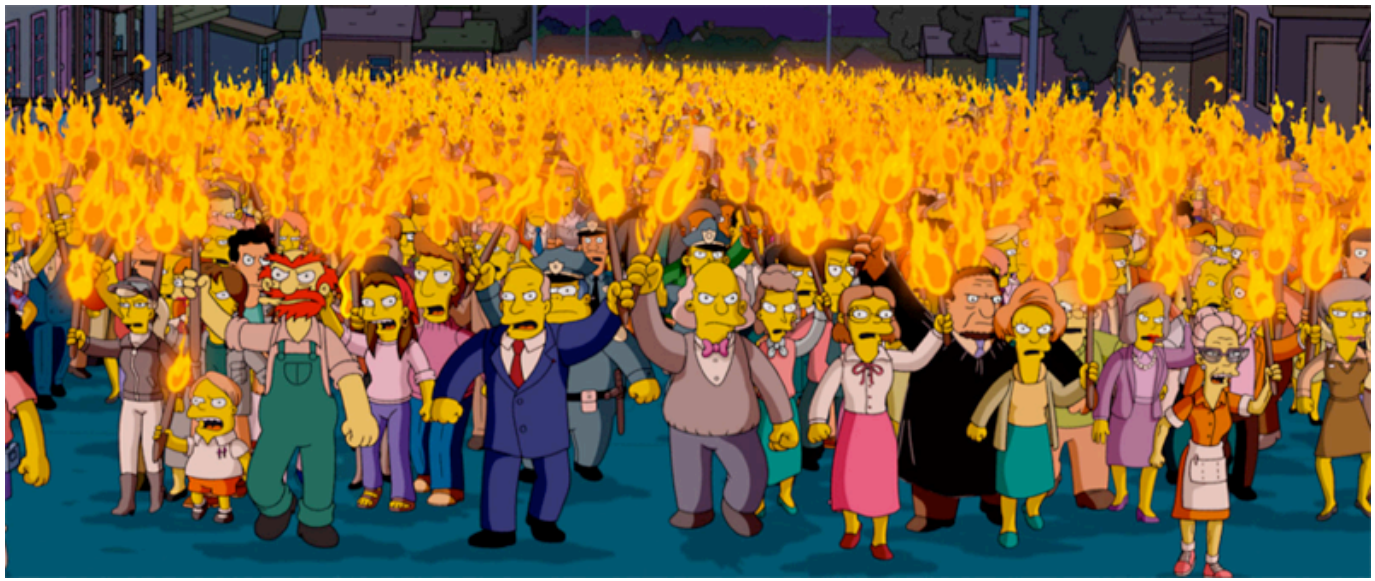
Not quite the remote I was thinking of (Ed.)

Summary

- Learned about Dependency Injection
- Saw how to make re-usable components
- Defined those objects in ColdSpring
- Created ColdSpring and Called your objects
- Got your AOP on
- Exposed yourself (as a web-service)

The cake is a lie...

Q and A



- You ask the Questions
- I shall answer them (If I can!)
- You can be a reformed Spaghetti coder too!
- mark@markdrew.co.uk