webqem

Adding Structure to ColdFusion Applications with FW/1

Marcin Szczepanski
webqem

Wednesday, 13 March 13

My name's Marcin Szczepanski and today I'm going to be talking to you about adding some structure to Coldfusion applications, specifically using FW/1. I'll give you a quick background on what frameworks are and how to when I think they should be used, and then we'll dive into the major features of FW/1.

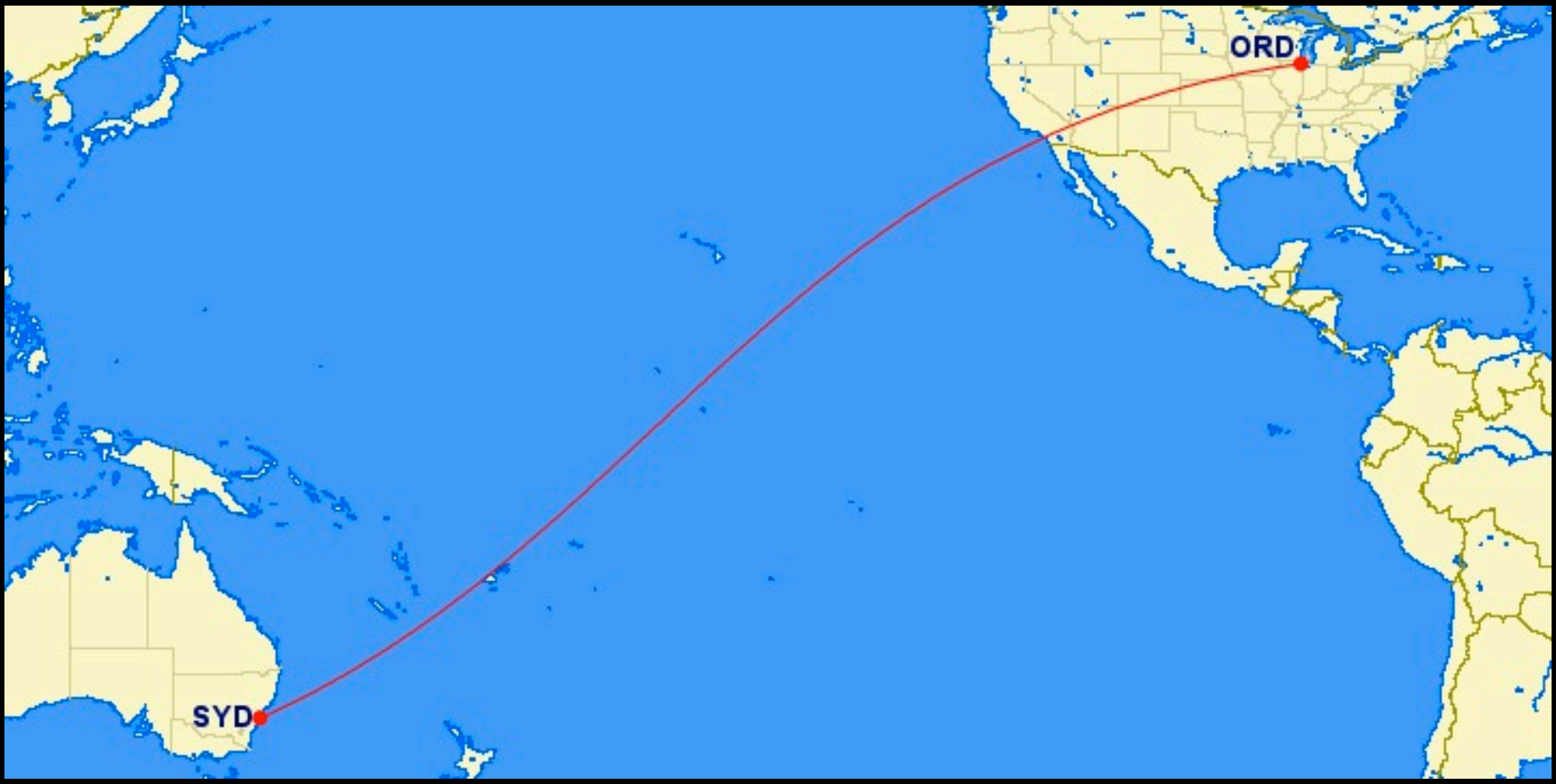
Me

webqem

Wednesday, 13 March 13

First a bit about me, just so you know who I am. My name's Marcin Szczepanski, I'm a web developer for a digital agency called webqem in Sydney, Australia. I was born in Poland but have lived in Australia since I was 4 years old.

Coming at you from 9,232mi (14,857km) away...



I think it's pretty awesome that I get to present to you all from half way around the world, thank you for the opportunity!

Lucas. Born three weeks ago.

Wednesday, 13 March 13

I apologise if this presentation isn't as polished as it could be, blame this little guy Lucas who came into our family just over three weeks ago. It wouldn't be a proper presentation without some baby photos!

Wednesday, 13 March 13

He's not always screaming like in that previous photo.

My first memories of Chicago

I've never been to Chicago, but I did spend a lot of time there as a youngster - starting with Microsoft Flight Simulator 4 and it's Meigs Field default.

Wednesday, 13 March 13

I work for webqem, a digital agency based in Sydney
I’m a "developer" - essentially covering everything from back-end to front-end, so Coldfusion is only a part of what I do.

I've been developing with ColdFusion for almost 6 years, basically from around the CF7 days, but have been involved with the web professionally since 1999.

FW/1

Wednesday, 13 March 13

I started using FW/1 about two years ago, as it seemed to be a nice change from some of the "heavier" CF frameworks out there like FuseBox, ColdBox, and so on.

What attracted me to this framework was it's simplicity - it's a single CFC that you drop into your application and it’s convention over configuration, so you need to do very little in order to get a simple app up and running using the framework.

Agenda

- What is MVC?
- What are frameworks?
- FW/1 Basics
- Dependency Injection
- FW/1 Handy Features

Wednesday, 13 March 13

So just to summarise what I'm going to cover today - this is going to be an introductory presentation, so I'm going to talk about what frameworks are at a high level, I'll briefly talk about what the term “MVC” is and how it relates to frameworks.

We’ll then get stuck into FW/1 and I’ll show you how FW/1 applications are put together and how they work.

I’ll talk about Dependency Injection which is kind of an advanced topic but I think it’s pretty easy to get started with and makes talking to your services a lot easier.

Last of all I’ll touch on a few handy features in FW/1

There'll be a few demos along the way, and all the code and these slides will be available on github after the presentation.

So we're going to cover a fair bit of material in a short time, so it might seem a bit daunting if you’re new to some of these concepts, but I wouldn't expect someone who had never used FW/1 to go out and start building apps with it straight after this presentation. Treat it as an overview of the major areas of the framework, which will at least help you get your bearings a little once you start using the framework yourself.

MVC

webqem

You might have heard the term “MVC” as it relates to web frameworks, MVC stands for “Model View Controller” and is a so called “Design Pattern” for building web applications. Most web frameworks out there are MVC or some variant thereof, including FW/1. I’ll cover the academic side of just what MVC is.

Model-View-Controller

- Model
- View
- Controller

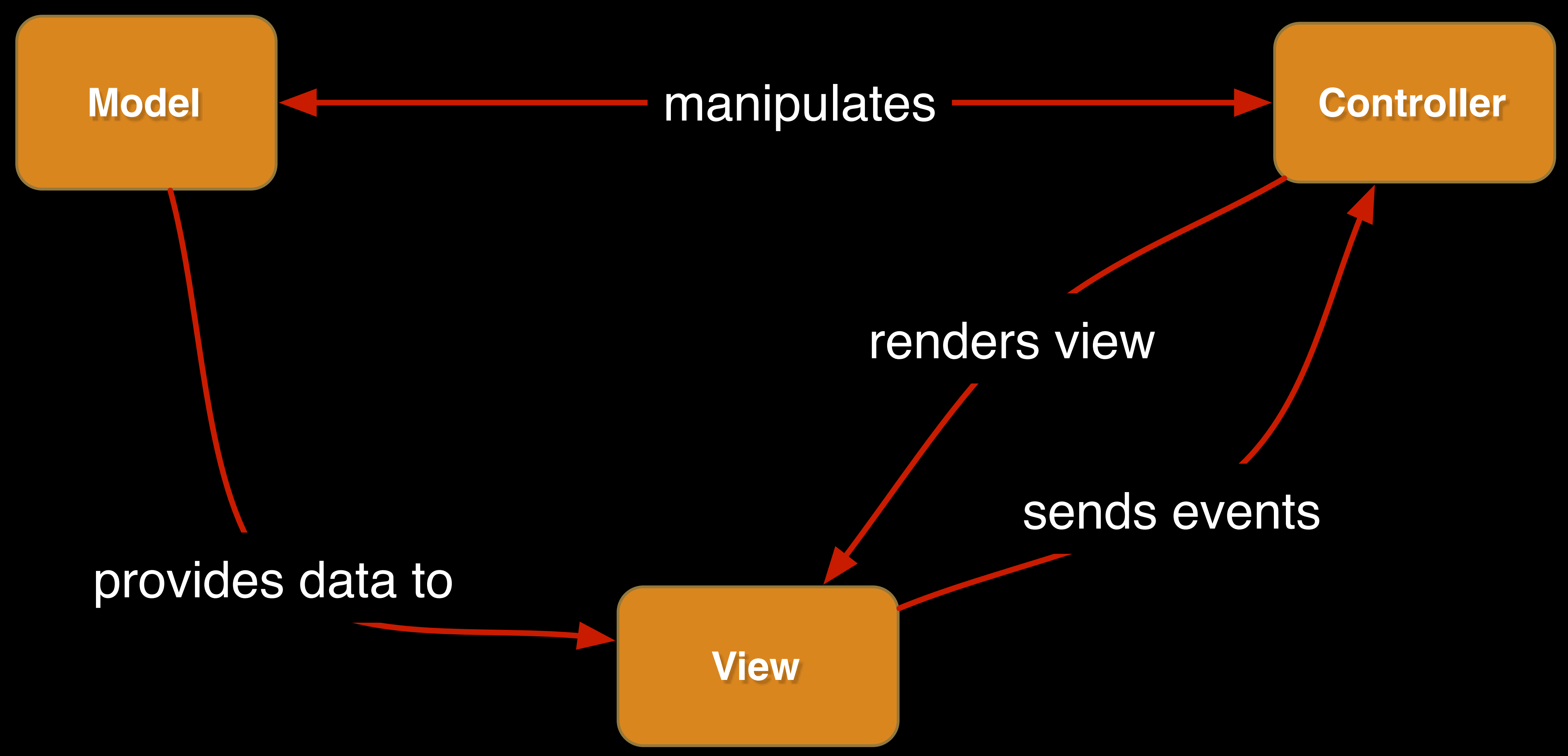
In a nutshell MVC is about separation of concerns for the different parts of your app - into, as the name implies, Model, View and Controller.

The Model is your data model / services - this is where your business logic lives - including interfacing with other systems/databases, the description of your data structures themselves, and so on.

The View is what the user sees - in a web app this is effectively the HTML that gets served to your user (although with things like ReST and JSON this is a simplification).

Finally the Controller is what brings the view and model together - the controller's job is to take the user input from the view, manipulate the model as a result of the input, and return updated views.

Model-View-Controller



webqem

Wednesday, 13 March 13

FW/1 effectively provides the Controller and View parts of your application, and has integration points for your Model but does not proscribe what that Model is.

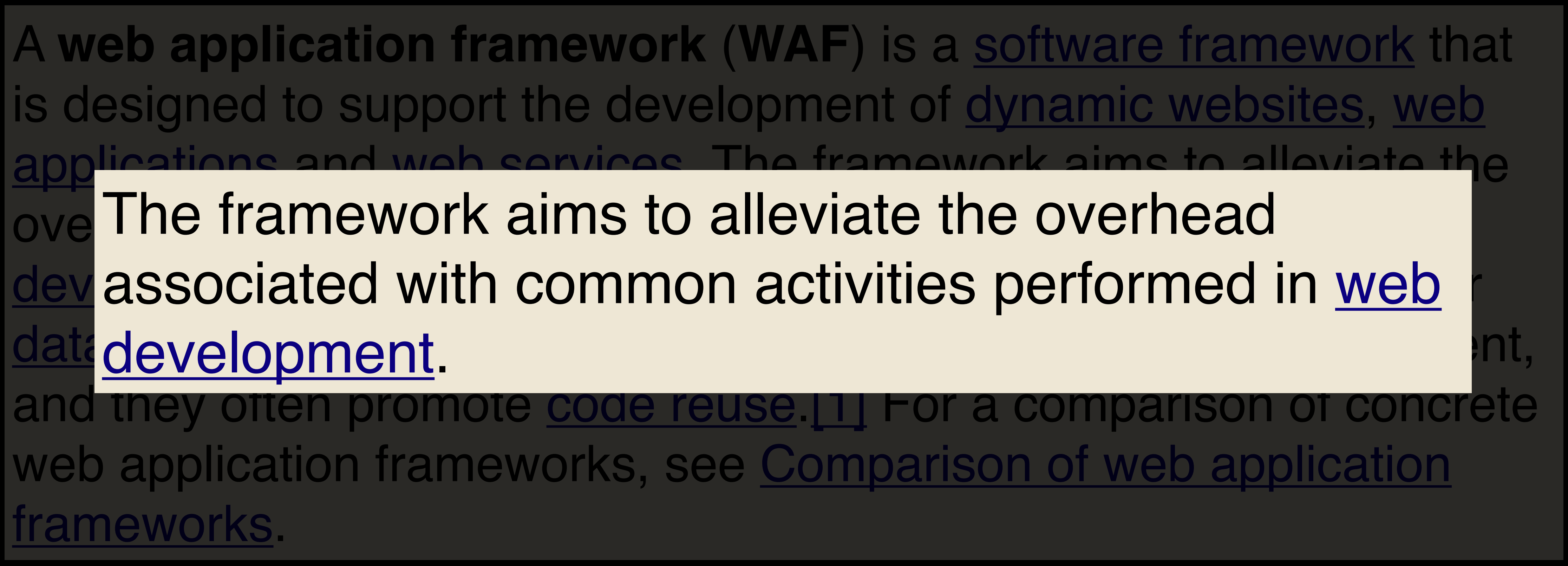
So as an example, say we have a HTML form that asks for your name - this form is contained inside a View. When you fill in the form and Submit it, the data is sent back to your Controller. So let's say our Controller has access to a service that makes the name you entered uppercase, it then switches the view to a "Hello" view and show your name in uppercase. This is a very simplified example, but as we get into FW/1 we'll see how the framework fits into this.

Frameworks

Wednesday, 13 March 13

So why do you need a framework?

What is a framework?



A **web application framework** (**WAF**) is a software framework that is designed to support the development of dynamic websites, web applications and web services. The framework aims to alleviate the

ove

dev r

data ent,

and they often promote code reuse.[1] For a comparison of concrete web application frameworks, see Comparison of web application frameworks.

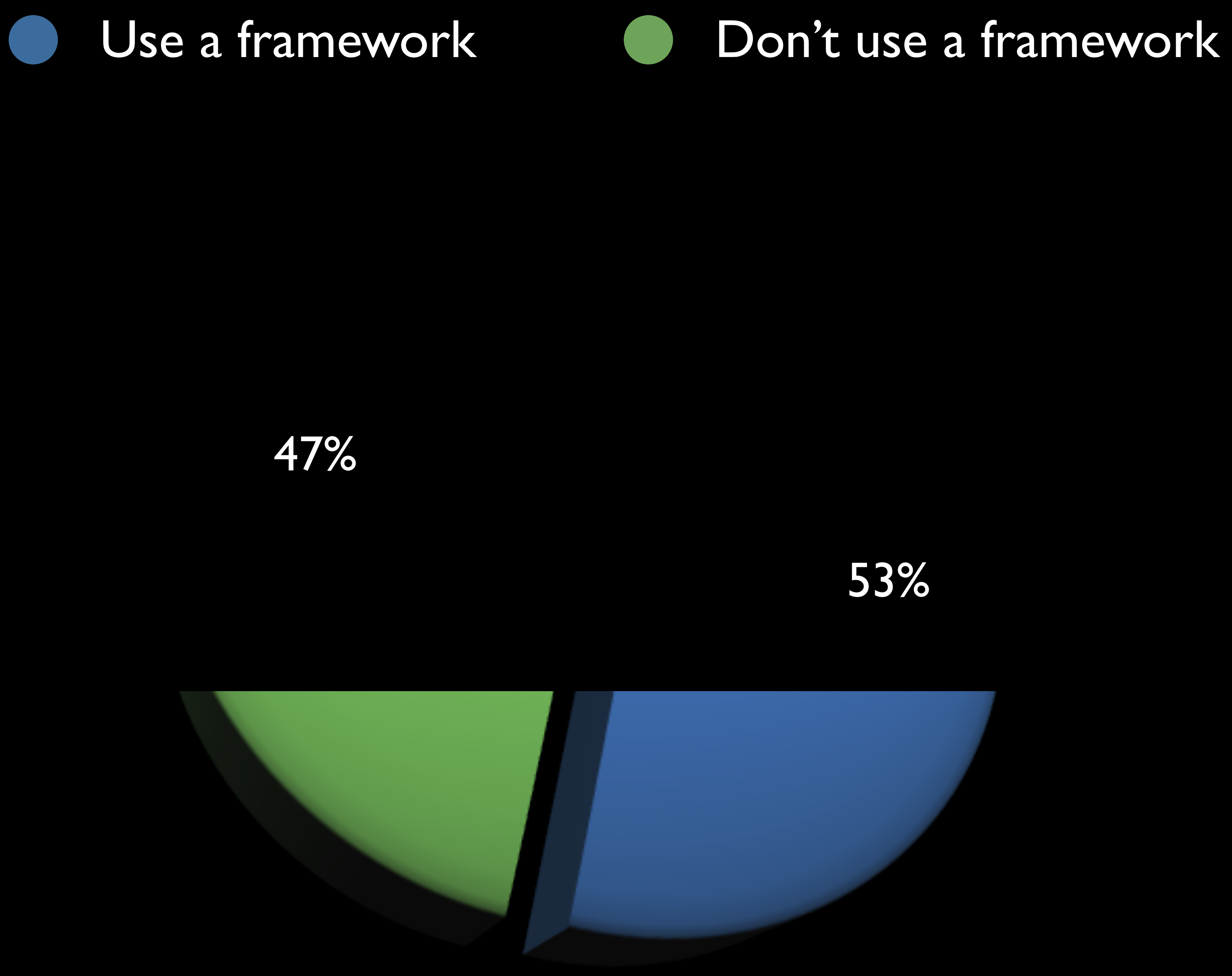
The framework aims to alleviate the overhead associated with common activities performed in web development.

Source: http://en.wikipedia.org/wiki/Web_application_framework

Wednesday, 13 March 13

So what is a framework? As the name implies, it's a standard base for your application. There are frameworks for all sorts of things, not just web applications. However in the context of a web application a framework helps you with the boilerplate plate code you would end up putting into every application, it helps with common scenarios and provides common solutions to problems that you're likely to have.

CF Framework Usage



Source: http://cfunited.com/blog/index.cfm/2013/2/12/State-of-the-CF-Union-survey-2013--results-and-winner

webqem

Wednesday, 13 March 13

In the recent CFUnited State of the CF Union survey 47% of people said they don't use a framework - unfortunately there wasn't a follow-up question as to "why not?", but given that 47.6% of people's biggest challenge is maintaining badly written code I'd say that probably a lot of people are working on maintaining old applications rather than building new ones.

Why use a framework?

- Adds structure
- Aids maintainability
- Provides plumbing
- Aids testing
- Adds a support group

So if you are starting a new application today, why should you use a framework?

A framework gives you structure - it means various parts of the application are in consistent places, which makes collaboration and maintenance easier. A framework gives you the plumbing so that you can focus on building the parts of your app that make your app unique, rather than dealing with the same boilerplate over and over.

Generally people run into the same common issues when building web applications and a framework should provided solutions for these sort of problems you're likely to run into.

Why wouldn't you use a framework?

- You wouldn't not use a framework! ;)

Wednesday, 13 March 13

My personal belief is there's very few apps where you shouldn't use some sort of framework. If you start without a framework you'll eventually end up either reinventing a framework, or just suffering through the problems that frameworks solve.

That said, if you're just building a simple one page "quick and dirty" application then there's no reason you can't just throw a CFM together for it. If it needs to grow there shouldn't be any issue with moving it into a framework down the track.

FW/1

Wednesday, 13 March 13

So let's talk about FW/1. Why FW/1?

Frameworks can vary in the amount of "weight" they introduce to your application. Now obviously this talk is about FW/1 so I'm going to be biased towards that, and I admit I haven't looked at the competition for a number of years, but in my previous experience with frameworks like ColdBox, Mach II, and Fusebox, I found them to be quite a "heavy" frameworks - the framework itself added a lot of overhead, it proscribed the way of doing a lot of things - including how you build your models and services, and generally still meant you had to write quite a bit of boilerplate.

Finding FW/1 those years ago was like a breath of fresh air, FW/1 is very light - it doesn't proscribe a lot of things, and requires very little FW/1 specific code.

FW/1

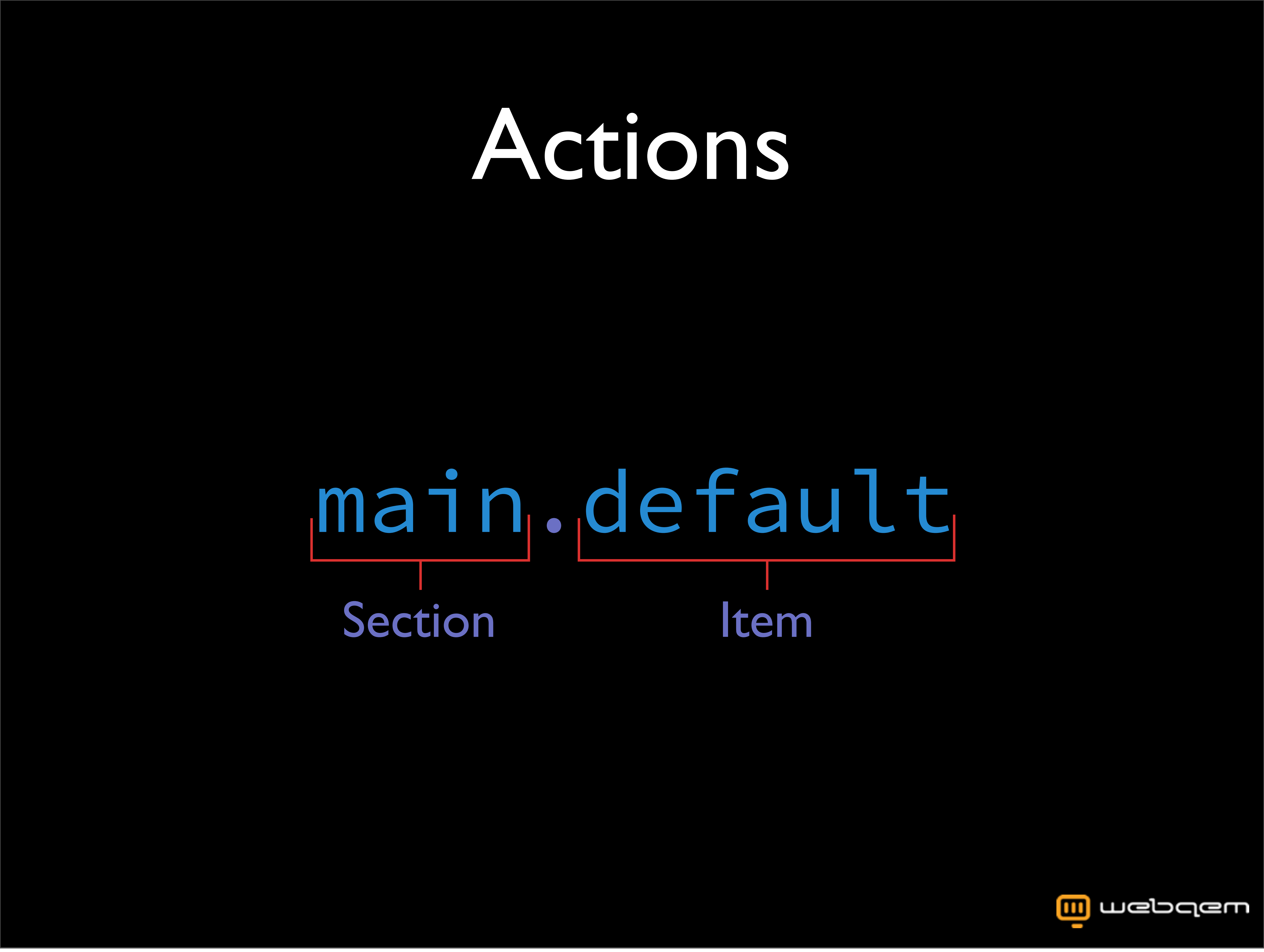
- Single CFC
  - `cfcomponent extends="org.corfield.framework" {`
- Convention over Configuration
- Configure via `variables.framework`

```
variables.framework = {
  reloadApplicationOnEveryRequest = true
};
```

Wednesday, 13 March 13

Why I like FW/1 is that it's very simple - it's a single CFC, and in order to make your application a FW/1 application you simple have your app.cfc extend the fw1.cfc.

FW/1 is all about "convention over configuration". It has sensible defaults for everything, so you only need to configure the framework for special cases rather than the norm. You configure the framework by setting values on the variables.framework structure in your app.cfc.



Actions

$$\underbrace{\texttt{main}}_{\text{Section}}.\underbrace{\texttt{default}}_{\text{Item}}$$

Wednesday, 13 March 13

You access your code in an FW/1 application via an "action"s - an example of an action name is "main.default" - it is comprised of a section and item name. This is actually FW/1's default action. Sections allow you to break your application up into logical components. We'll talk more about how actions map to actual controller code a bit later.

Views

views / *section* / *item.cfm*

views / main / default.cfm

Wednesday, 13 March 13

Each action in an FW/1 application has a corresponding view. A view is a CFM page containing the template for the output of an action.

Views are CFM pages named the same as the item, and they sit in a subdirectory with the name of the section, within the views directory.

A view is the only required part of a FW/1 application, but without a view there's nothing to render so the framework throws an error.

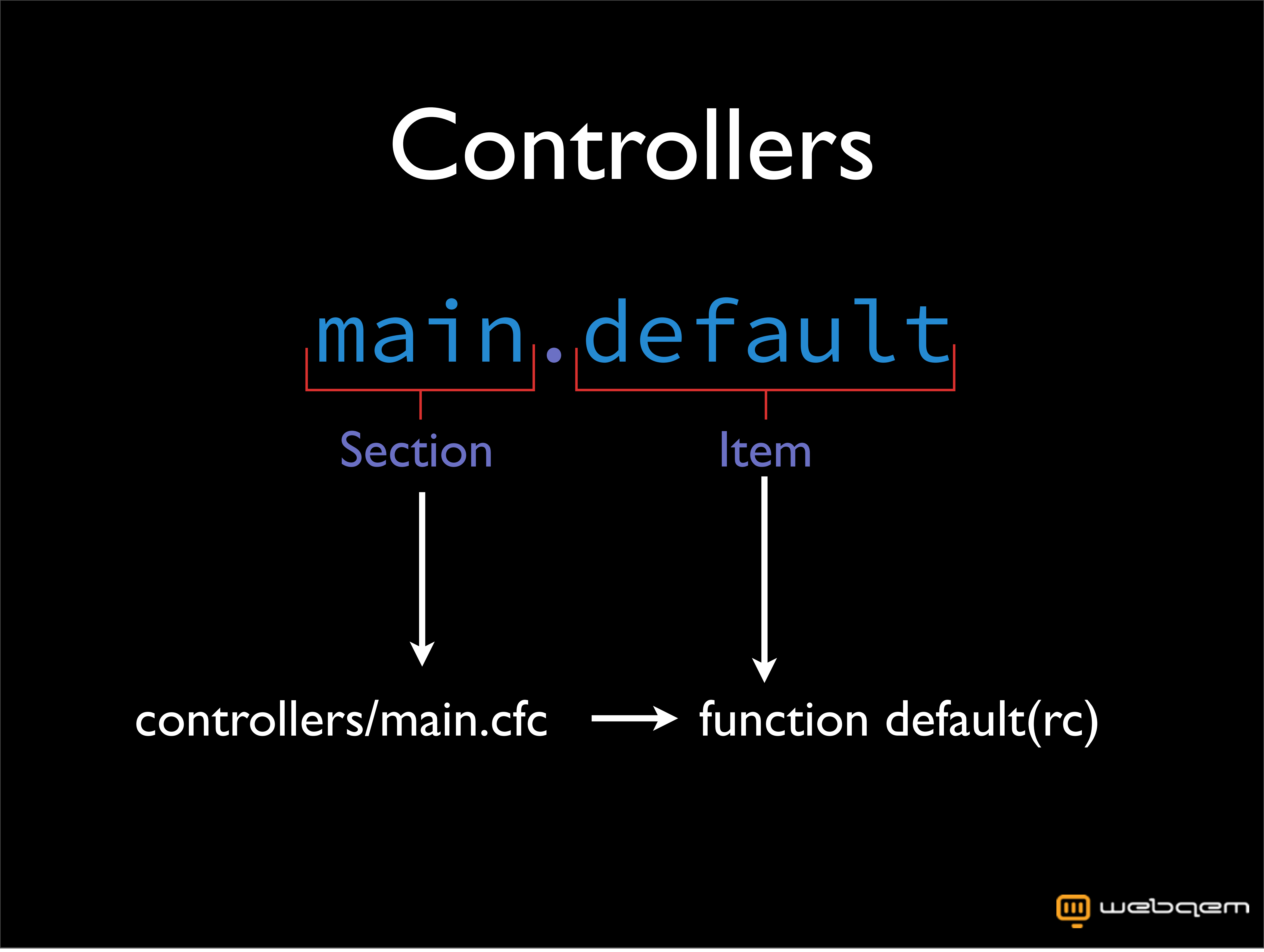
With the default configuration all a minimal FW/1 app requires a single view in views/main/default.cfm.

Minimal FW/1 Application

DEMO

Wednesday, 13 March 13

So let’s have a look at the structure of a minimal FW/1 application...



Controllers

$$\underbrace{\texttt{main}}_{\text{Section}}.\underbrace{\texttt{default}}_{\text{Item}}$$

$$\begin{tikzcd}\text{Section} \arrow[d] & \text{Item} \arrow[d] \\ \text{controllers/main.cfc} \arrow[r] & \text{function default(rc)}\end{tikzcd}$$

Wednesday, 13 March 13

Of course a minimal application like you just saw isn’t very useful – there needs to be somewhere to run your logic.

Now you could just go and add all your application logic into the view CFMs and build an application, but that would violate the MVC principle – where a view is just supposed to display the model and not manipulate it. It's the job of a controller to talk to the model, manipulate it's state and return the required model data to the view for presentation to the user.

So let’s have a look at how FW/1 achieves this.

When you call an action, such as "main.default" FW/1s convention over configuration approach means the framework will look for a controller component called main.cfc in the directory "controllers" – it retrieves or creates an instance of this controller, and then calls a function in this controller called "default", passing it what is called the "request context" or rc for short.

Controllers
Request Context

controllers/main.cfc

```
component output="false"
{
   public void function default(required struct rc)
   {
      rc.upperCaseName = UCase(rc.name);
   }
}
```

views/main/default.cfm

```
<p><strong>Hello <cfoutput>#rc.upperCaseName#</cfoutput>!</strong></p>
```

webqem

Wednesday, 13 March 13

The request context is a struct that contains any form and url parameters for the current request. It's the job of your controller method to use this information, plus other relevant information from the session or cookies, in order to get the required data from the model. The data the view will need is then also set in this rc struct.

You don't need to return anything from your controller, just add any required data into rc.

The request context will be available to your view for outputting any results.

Also take note that unlike some other frameworks there's nothing special about a FW/1 controller – it doesn't have to extend a particular class, it's just a regular Coldfusion Component. This, amongst other things, makes testing controllers a bit easier.

Now we haven't mentioned how to access your model / services yet, we'll talk about that shortly as it's probably the most complex part of all this – and there are a couple of ways to do it.

Layouts

Wednesday, 13 March 13

We've talked about views and how they relate to controllers, and before we talk more about how to build apps with these components we need to talk about layouts. Layouts are a part of the "view" system, and are designed to let you create views that adhere to the Don't Repeat Yourself principle by moving common or boilerplate code out of your individual views.

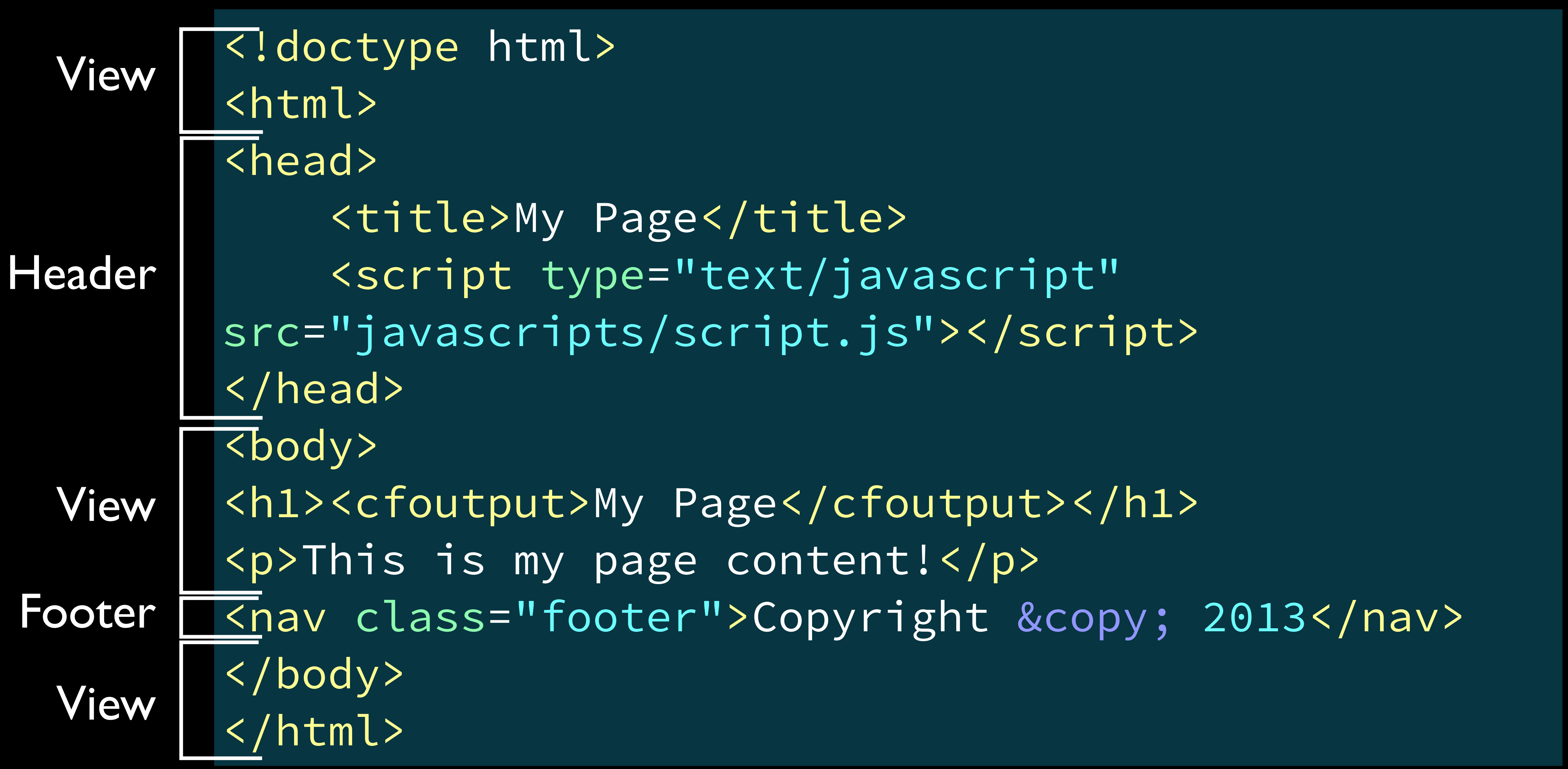
CFM based application

```
<!doctype html>
<html>
<cfset title = "My Page">
<cfinclude template="includes/header.cfm">
<h1><cfoutput>#title#</cfoutput></h1>
<p>This is my page content!</p>
<cfinclude template="includes/foooter.cfm">
</html>
```

webqem

You might have previously written CFM based applications that have views that look like this, which is rendered progressively.

Legacy Layout



```
<!doctype html>
<html>
<head>
    <title>My Page</title>
    <script type="text/javascript"
src="javascripts/script.js"></script>
</head>
<body>
<h1><cfoutput>My Page</cfoutput></h1>
<p>This is my page content!</p>
<nav class="footer">Copyright &copy; 2013</nav>
</body>
</html>
```

As you can see from the output the source of a particular piece of HTML jumps between the various components.

However the FW/1 view / layout system (and a lot of other web frameworks) is based on rendering the view from the inside out.

Layouts

layouts / default.cfm

```
<cfoutput>#body#</cfoutput>
```

webqem

At the simplest level you can have a single layout in layouts/default.cfm - before the layout is rendered the view is rendered into a variable called “body” that’s accessible to your layout, and then your layout is rendered. So your layout needs to output this special variable "body" which will include your view content. So basically you'd put all your HTML boilerplate, head tags, footers in your layout, and then inside the <body> tags you'd cfoutput body.

This means your view.cfm's remain focussed on the specific content related to that action, and not cluttered with the generic stuff.

Layouts

layouts / *section* / *item.cfm*

layouts / *section.cfm*

layouts / default.cfm

```
<cfset request.layout = false>
```

Now layouts themselves can also be nested - you can have section specific and even item specific layouts and again they will be rendered from the inside out.

In each layout you have to output the “body” variable to ensure that everything rendered to that point is output.

The other useful bit about the layout engine is that any time in your views or layouts you can do "request.layout = false" - this stops FW/1 from going further up the hierarchy when rendering. For example if your view is outputting plain text and you don't want the HTML boilerplate around it.

Why might you want an item specific layout, isn't that what the view is? Well in your controller you can override the view that renders a particular action, but you might still want some piece of layout specific to that item regardless of which view gets rendered.

CFM based application

index.cfm

```
<!doctype html>
<html>
<cfset title = "My Page">
<cfinclude template="includes/header.cfm">
<h1><cfoutput>#title#</cfoutput></h1>
<p>This is my page content!</p>
<cfinclude template="includes/foooter.cfm">
</html>
```

webqem

Wednesday, 13 March 13

You might have previously written CFM based applications that have views that look like this, which is rendered progressively.

Layouts & Views

views/main/default.cfm

```
<cfoutput>
    <h1>#rc.title#</h1>
    <p>This is my page content!</p>
</cfoutput>
```

layouts/main.cfm

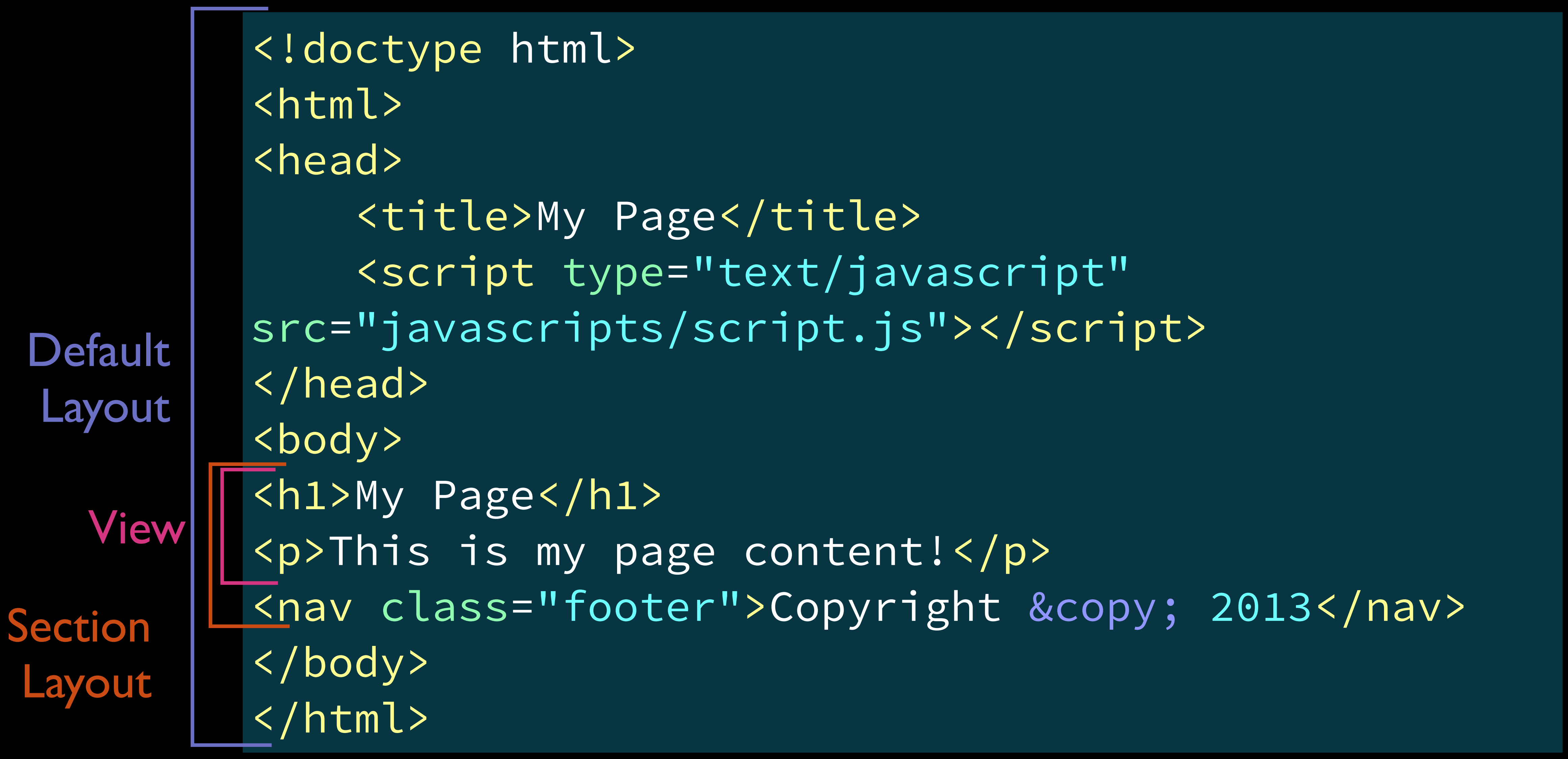
```
<cfoutput>#body#</cfoutput>
<nav class="footer">Copyright &copy; 2013</nav>
```

layouts/default.cfm

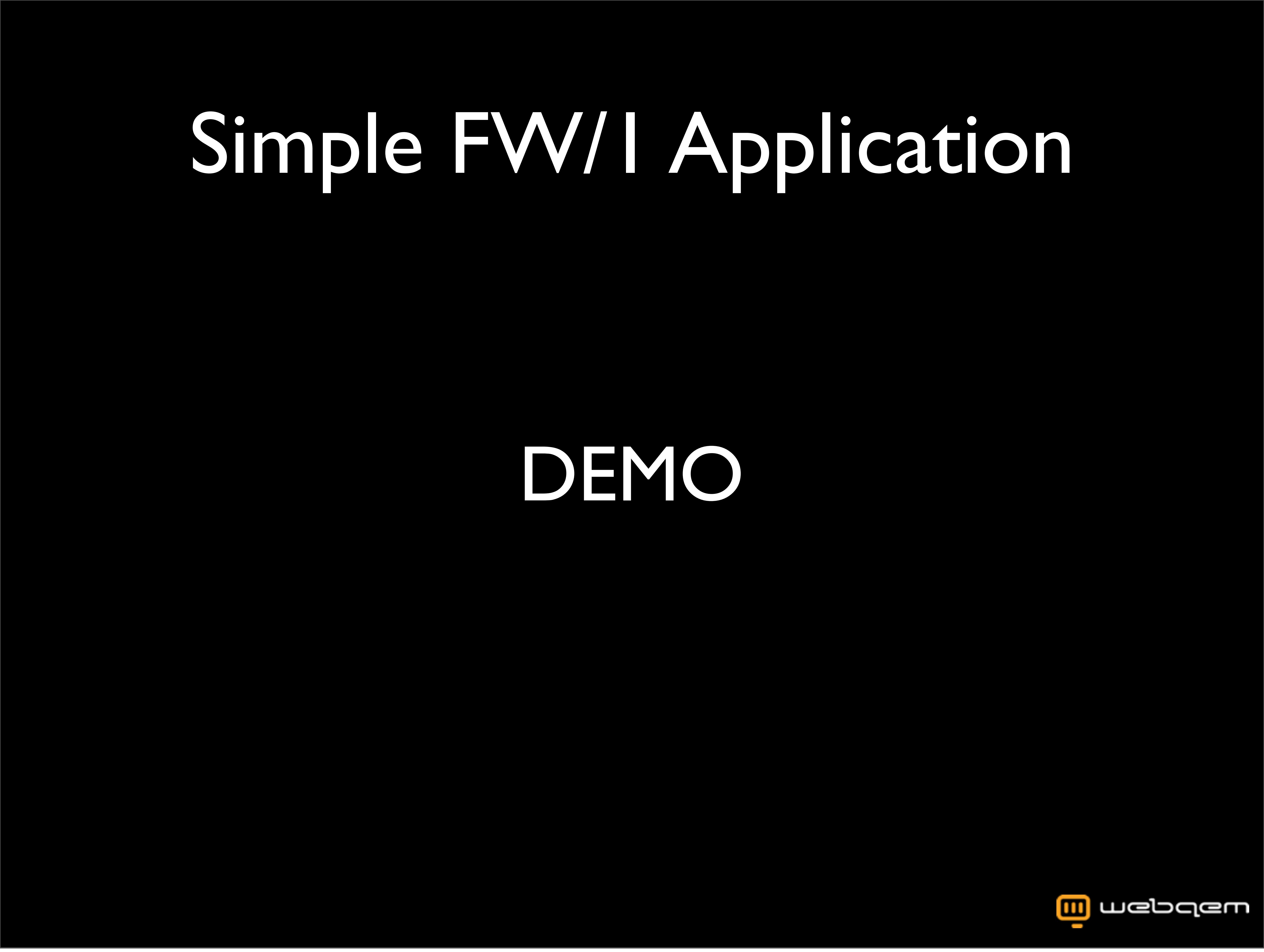
```
<!doctype html>
<html>
<head>
    <title><cfoutput>#rc.title#</cfoutput></title>
    <script type="text/javascript" src="javascripts/script.js"></script>
</head>
<body><cfoutput>#body#</body>
</html>
```

webqem

FW/1 Nested Layout



```
<!doctype html>
<html>
<head>
    <title>My Page</title>
    <script type="text/javascript"
src="javascripts/script.js"></script>
</head>
<body>
<h1>My Page</h1>
<p>This is my page content!</p>
<nav class="footer">Copyright &copy; 2013</nav>
</body>
</html>
```



Simple FW/1 Application

DEMO

Wednesday, 13 March 13

So let's have a look how this all fits together with a demo of a more complete FW/1 application.

Services and your model

Wednesday, 13 March 13

So now you have an idea of the bit of FW/1 that make FW/1 FW/1 - the view and controller system - let's talk about services and the model. The approach to services in FW/1 has evolved over the versions, and I won't cover the history here, but talk about two approaches to calling services.

Services

- Contain your business logic
- Talk to other systems - databases, APIs, etc
- Should have no knowledge of FW/1 / the HTTP Request

The CFCs you call as services ideally should have no knowledge of FW/1 or even HTTP - your controller should prepare any parameters the service needs and call the service - you shouldn't be accessing the session or cookie scope, and you shouldn't just pass "rc" as a parameter to your service. This kind of coupling makes things harder to test.

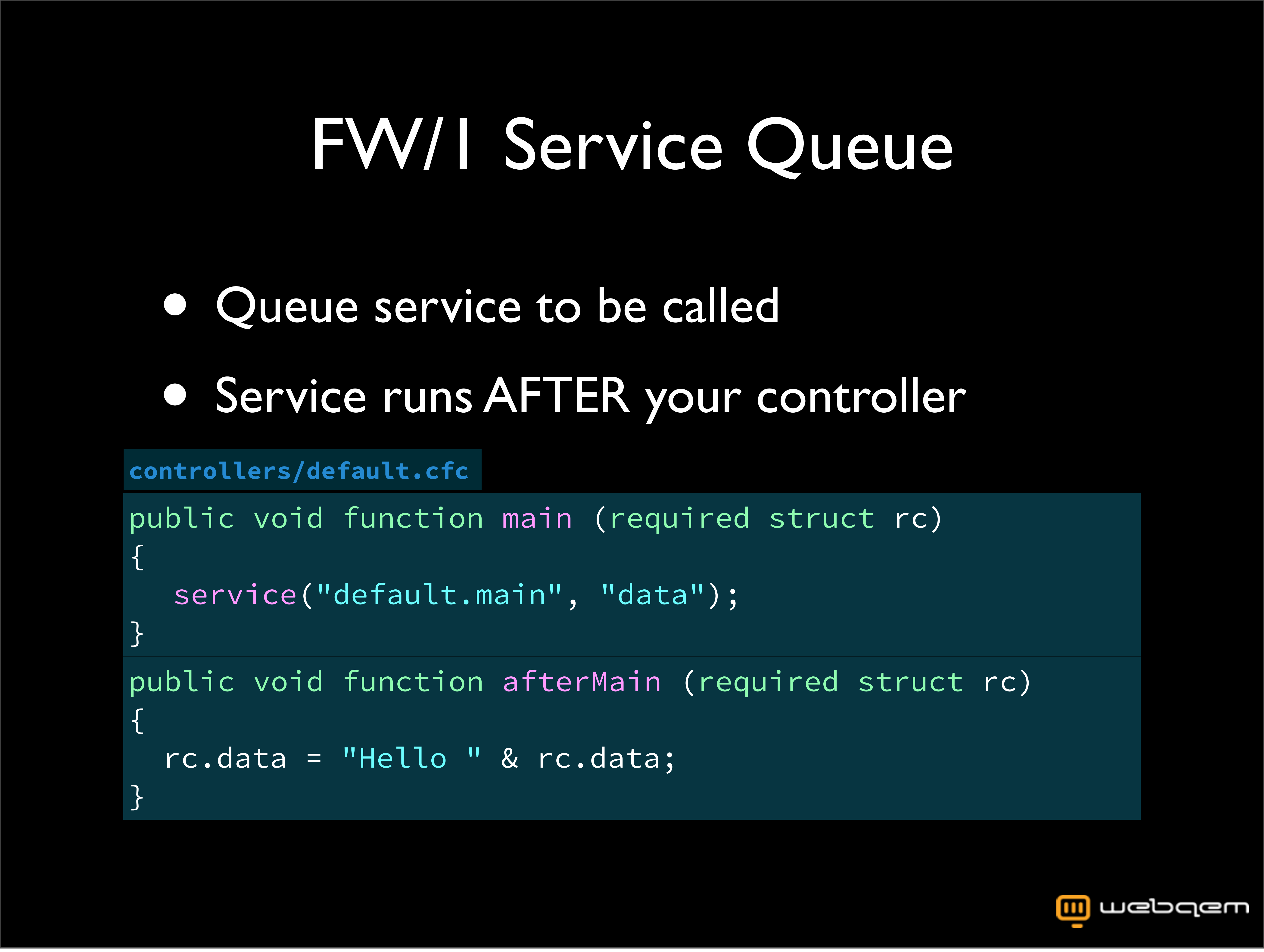
The Service Queue

- Standard approach to calling services in FW/1
- Might be deprecated in the next major version

Wednesday, 13 March 13

So there's the concept of a service queue in FW/1, and if you read the documentation this is the approach talked about when calling services. However it's always been a source of confusion for new users due to some quirks in the way it has been designed and works, and it's use has evolved over the various framework versions.

Recently Sean Corfield has mentioned on the mailing list that it probably isn't the best way of calling services, and is likely to be deprecated in future versions in favour of calling services directly.



FW/1 Service Queue

- Queue service to be called
- Service runs AFTER your controller

controllers/default.cfc

```
public void function main (required struct rc)
{
   service("default.main", "data");
}
public void function afterMain (required struct rc)
{
  rc.data = "Hello " & rc.data;
}
```

Wednesday, 13 March 13

FW/1 has a service() function that allows you to queue a service call at the end of your controller method. This uses the same convention over configuration approach to locate and call your service function.

Now this is the key thing that is a trap for young players – the service call is queued and not executed immediately. It is executed after your controller method has run! So if you need to manipulate the result of your service call before you render the view, you'll need to split your controller up around the service call.

Sidebar: Request Lifecycle

```
application.cfc::before
controller.cfc::before
controller.cfc::startItem
controller.cfc::item
```

Call queued services

```
controller.cfc::endItem
controller.cfc::after
application.cfc::after
```

webqem

Wednesday, 13 March 13

There’s actually more than just your single controller method called during the request lifecycle - in actuality there are a number of functions that will be called if they exist, at various levels of granularity. For example the before and after functions in app.cfc get called for every request in the application.

You can use these to implement things such as authentication, some sort of custom logging, etc.

FW/1 Service Queue

controllers/default.cfc

```
public void function main (required struct rc)
{
   service("default.main", "data");
}
public void function afterMain (required struct rc)
{
  rc.data = "Hello " & rc.data;
}
```

webqem

Wednesday, 13 March 13

FW/1 has a service() function that allows you to queue a service call at the end of your controller method. This uses the same convention over configuration approach to locate and call your service function.

Now this is the key thing that is a trap for young players – the service call is queued and not executed immediately. It is executed after your controller method has run! So if you need to manipulate the result of your service call before you render the view, you'll need to split your controller up around the service call.

Dependency Injection

Wednesday, 13 March 13

So let's talk about "Dependency Injection", what it is and how it can make calling services, reduces coupling and in general fits in with the tenet of having well structured applications. It might not seem simple initially, but really at it's core it is, and once you understand it you'll wonder how you ever lived without it!

The Classic Way

controllers/main.cfc

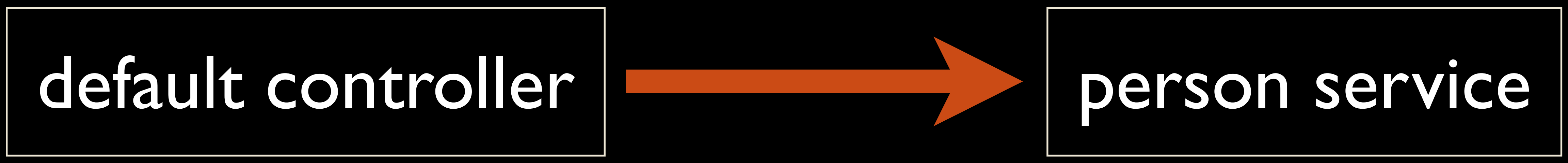
```
component output="false" accessors="true"
{
    property name="personService";

    public any function init ()
    {
        var service = createObject("component", "model.service.person").init();
        setPersonService(service);
        return this;
    }

    public void function default (required struct rc)
    {
        rc.qPeople = getPersonService().getPeople();
    }
}
```

Wednesday, 13 March 13

So at the minimum, there's nothing wrong with instanciating CFCs in your controller and calling the methods in those CFCs like you might do in your regular non-framework applications. FW/1 is not at all proscriptive as to how this is done.

If you're using ORM and it's a simple app you might be able to just make your ORM calls directly in the controller - sure this might be coupling things a little tightly but it might be acceptable depending on your requirements - we build a lot of small promo / microsites where there's not really a point over-architecting for something that'll only be running for a few weeks.

Dependency Injection

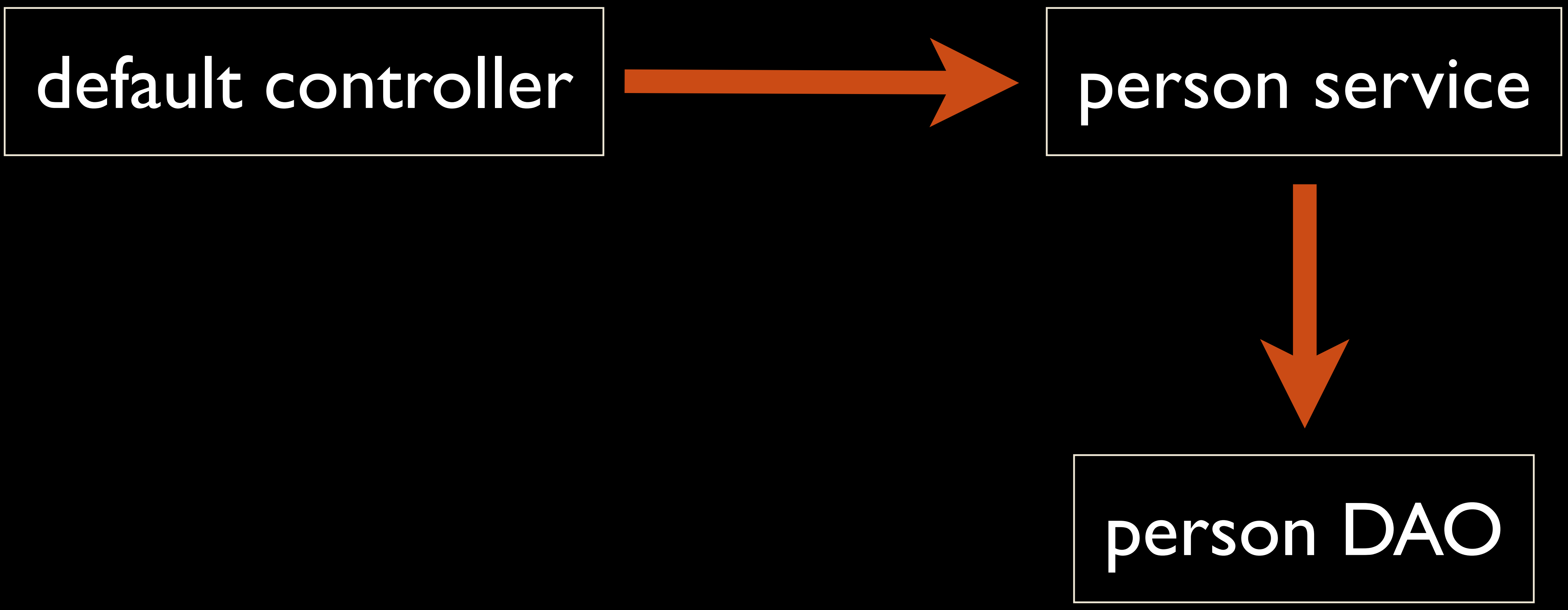


Wednesday, 13 March 13

Let’s say we have our default controller, and we’d like to use a “Person Service” to get some data about people to display to the user.

This means that the person service is now a dependency for the controller, and we need some way to provide the default controller with a person service.

Dependency Injection



Now let’s say that the person service requires a DAO to actually perform the data fetch from an external data store of some sort. This now means that person service has a dependency of it’s own – the person Data Access Object.

Dependency Injection

controllers/main.cfc

```
component output="false" accessors="true"
{
    property name="personService";

    public any function init ()
    {
        var dao = createObject("component", "model.dao.person").init();
        var service = createObject("component", "model.service.person").init(dao);
        setPersonService(service);
        return this;
    }

    public void function default (required struct rc)
    {
        rc.qPeople = getPersonService().getPeople();
    }
}
```

webqem

Wednesday, 13 March 13

This means that in our constructor we now need to do something like this, initialise the personDAO, initialise the personService passing it the personDAO, and finally set our personService.

You could also have the personService initialise it's own personDAO, but the end result is the same – you're tightly coupling the controller, service and DAO. If I wanted to change the component that's used as the DAO I'd need to modify my controller or service.

So what does this look like with a dependency injection framework?

Dependency Injection

controllers/main.cfc

```
component output="false" accessors="true"
{
    property name="personService";

    public void function default (required struct rc)
    {
        rc.qPeople = getPersonService().getPeople();
    }
}
```

Wednesday, 13 March 13

.. like this. Your controller no longer needs to concern itself with where the personService comes from, it can just assume that it's there. The Dependency Injection framework handles resolving the dependencies.

Bean Factory

- any getBean (string bean)
- boolean containsBean (string bean)

Wednesday, 13 March 13

- DI framework provides a "Bean Factory"
- Bean is just a java object
- Bean Factory is any component that implements these methods

A Dependency Injection framework provides what is called a Bean Factory. The name "bean" comes from the Java world (Java Bean - get it?) - it effectively refers to a component that contains public getters and setters for it's properties - for our purposes we just need to know that a "bean" is a component. So the responsibility of the bean factory, as the name implies, is to produce beans or components.

At it's core a Dependency Injection framework, at least those used by FW/1, needs to provide two functions - getBean and containsBean. Both take a bean name as a string, getBean will return some an instanciated object representing the bean asked for (if it exists).

FW/1 Bean Factory

Application.cfc

```
public void function setupApplication ()
{
    var beanFactory = ...;
    setBeanFactory(beanFactory);
}
```

Wednesday, 13 March 13

- bean factory set in setupApplication

You tell FW/1 about your bean factory by using the setBeanFactory function, here you can see how you would set a bean factory in your setupApplication function, we’ll get to the specifics of creating the bean factory shortly.

DI/1

Wednesday, 13 March 13

The most popular DI framework in Coldfusion is still Coldspring, but I'm not going to talk about that here because it's too complex for our purposes, I'm going to talk about Sean Corfield's DI/1. Chances are you don't need all the features in CS, and DI/1 will more that be enough for you. Sean has also hinted that he plans on making DI/1 part of the FW/1 distribution so it'll be even easier to setup.

Like FW/1 DI/1 is all about convention over configuration, we only need to add a few lines to app.cfc to start using DI/1.

FW/1 Bean Factory

Application.cfc

```
public void function setupApplication ()
{
    var beanFactory = new ioc("./model");
    setBeanFactory(beanFactory);
}
```

Wednesday, 13 March 13

So to use DI/1 you create an instance of the ioc component, the DI/1 implementation, and pass it the path to you service components. You then use setBeanFactory to set the FW/1 bean factory.

Now, I'll show you by way of a demo, but basically when a controller CFC is instanciated then FW/1 will check any properties for matching beans that it knows about, and will "inject" instances of those beans into the controller for you.

DI/1

DEMO

Wednesday, 13 March 13

DI/1 demo.

Recap

- Views
- Layouts
- Controllers
- Services

Wednesday, 13 March 13

OK, so we now know how a FW/1 application is structured - we know how to create views and layouts, how to create our controllers, and how to call our services.

Essentially these are all the building blocks you need to build any application.

To wrap up I'm now going to go over a few other "helper" features that FW/1 provides that you might find handy in day-to-day use. All these features are well documented in the wiki on Github, so if you want more detail on any of these, or in fact any of material already covered, then definitely "RTFM" as it's a very thorough manual.

Useful FW/1 Functions

Configuration

```
variables.framework = {
  action = 'action',
  usingSubsystems = false,
  defaultSubsystem = 'home',
  defaultSection = 'main',
  defaultItem = 'default',
  subsystemDelimiter = ':',
  siteWideLayoutSubsystem = 'common',
  home = 'main.default', // defaultSection & '.' & defaultItem
  // or: defaultSubsystem & subsystemDelimiter & defaultSection & '.' & defaultItem
  error = 'main.error', // defaultSection & '.error'
  // or: defaultSubsystem & subsystemDelimiter & defaultSection & '.error'
  reload = 'reload',
  password = 'true',
  reloadApplicationOnEveryRequest = false,
  generateSES = false,
  SESOmitIndex = false,
  unhandledExtensions = 'cfc',
  unhandledPaths = '/flex2gateway',
  trace = false
};
```

Source: https://github.com/seancorfield/fw1/wiki/Developing-Applications-Manual

One thing we didn't only glossed over was the FW/1 configuration. Using the variables.framework structure in app.cfc you can override all of the conventions in FW/1 - for example if you wanted to make the default action "site.index" you can change the defaultSection and defaultItem options. This is also where you configure whether FW/1 should use so called "search engine safe" URLs, and so on, which leads us to...

buildUrl

- Used to generate URLs to FW/1 action

```
buildUrl(action="customers.view", queryString="id=5")
```

```
/index.cfm/?action=customers.view&id=5
/index.cfm/customers/view/id/5
/customers/view/id/5
/myapp/customers/view/id/5
```

```
<cfoutput>
  <a href="#buildUrl(action="customers.view", queryString="id=5")#">View Customer</a>
</cfoutput>
```

Don't forget to **cfoutput**!

webqem

This is probably one of the most common FW/1 funcitons you'll use when building up your views and layouts.

In your views, if you want to link to a FW/1 action you can use the buildUrl function which will generate a proper URL to that action based on your configuration - of whether to use search engine safe URLs, use "index.cfm", any base paths and so on. It has a whole lot of ways you can use it, but at it's core most of the time you will use this syntax:

Accessing FW/1 in your controller

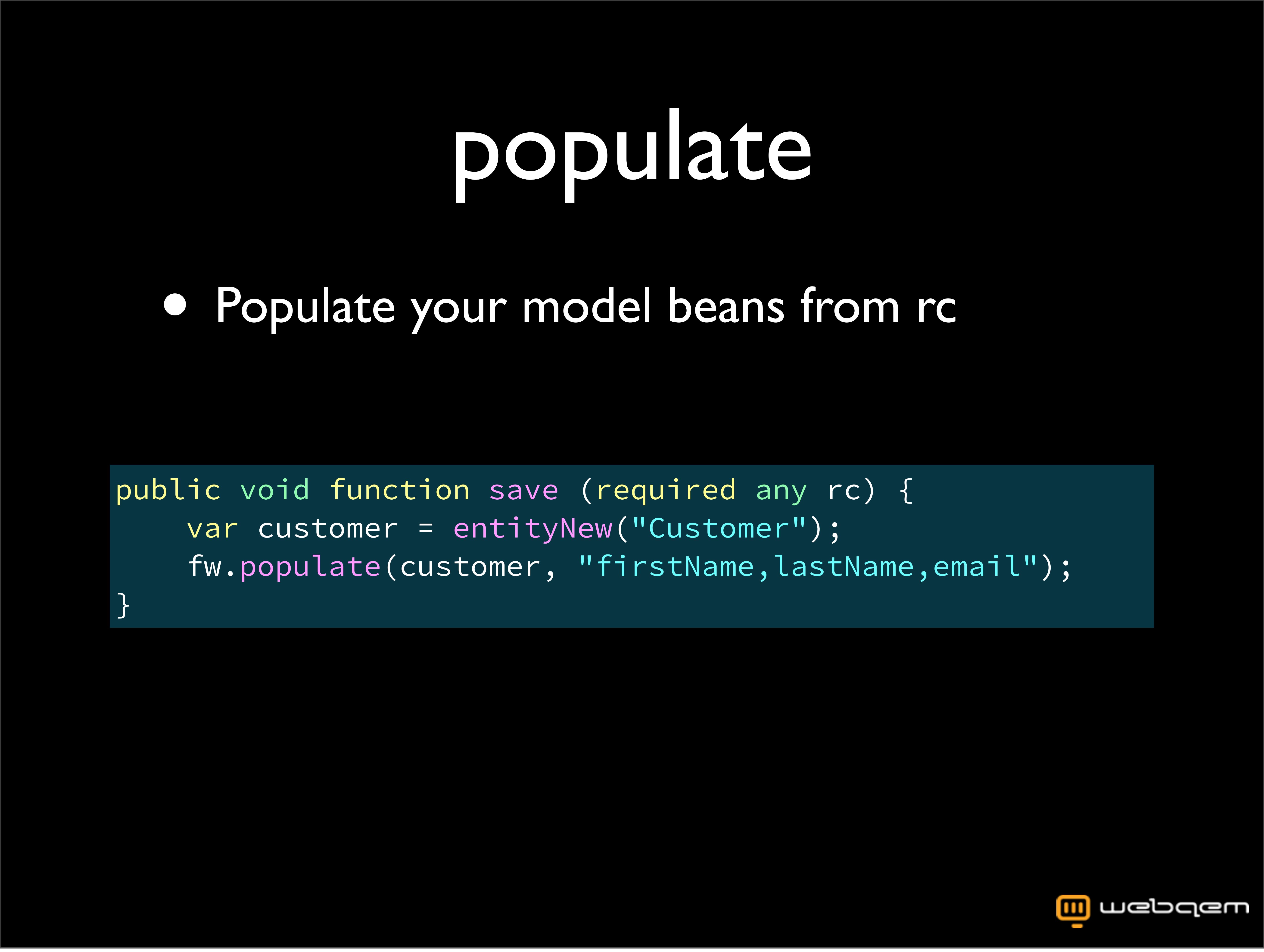
```
public any function init (required any fw) {
    variables.fw = arguments.fw;
    return this;
}
```

webqem

Wednesday, 13 March 13

So in order to access some of the other FW/1 functions in your controller you'll need a reference to the framework. Conveniently when FW/1 instanciates your controller for the first time it passes in a reference to itself to your constructor. The general pattern is to save this into the variables scope so you can access it later in your code.

This is necessary because a FW/1 controller is not a sub-class of some sort of framework controller like with other frameworks.



populate

- Populate your model beans from rc

```
public void function save (required any rc) {
    var customer = entityNew("Customer");
    fw.populate(customer, "firstName,lastName,email");
}
```

Wednesday, 13 March 13

The populate function is used to set properties on model components from your request context. This is useful in actions where you submit forms, for example.

It takes as it’s parameters the component you want to populate, and a “white-list” of attributes that are allowed to be set. This is to prevent issues that could occur if say your “user” object had an “is_admin” property, and a malicious user submitted a profile change and added an “is_admin” field to the submission data. Without a whitelist of allowed values this would happily overwrite that field and could lead to a security comprimise.

There are also some other parameters around trimming the values.

redirect

- Perform a HTTP redirect to a new action
- Optionally preserve rc keys using session scope

default.cfc

```
public void function save (required any rc) {
    var customer = entityNew("Customer");
    fw.populate(customer, "firstName,lastName,email");
    if ( ! customer.valid()) {
        rc.errors = customer.getErrors();
        fw.redirect(action="customer.edit", preserve="errors" );
        return;
    }
}
```

webqem

In your controller you might want to redirect to a different action - note that this does an actual HTTP redirect, so a new request is made. the redirect function supports "preserving" RC keys using the session scope. Similarly to buildUrl it will obey your configuration in terms of the urls it creates.

setView / setLayout

- Allows overriding the view or layout for this request

default.cfc

```
public void function view (required any rc) {
    param name="rc.id" default="0";
    param name="rc.format" default="html";

    var customer = entityLoadByPK("Customer", rc.id);

    if (rc.format == "json") {
        rc.json = serializeJSON(customer);
        setView("default.json");
    } else if (rc.format == "print") {
        setLayout("print");
    }
}
```

webqem

As we've seen, by default the layout and view used are based on the current action, however you can override either the view or layout in your controller.

One great example of where this is handy is where you want to output different content types depending on what was requested. For example to output JSON we don't want a layout, and we want to set a different content type. The way I like to do this is have a single "json" view that will output the contents of rc.json as JSON.

You can either override the view in individual controller methods, or you can get a bit more generic and do this at the application level in the "after" function.

Routes

- Allow you to map "friendly" URLs to FW/1 actions

Application.cfc

```
variables.framework.routes = [
    {"/customer/:id" = "/customer/view/id/:id"}
];
```

- buildUrl / redirect won't create a "routed" URL, only the canonical URL!

webqem

Wednesday, 13 March 13

Routes can be a whole presentation all in itself, and I cover routes in a lot of detail in my advanced presentation, however in their simplest form you can use routes to have action URLs like this instead of this. You configure routes in app.cfc .. example ..

Demo

Agenda

- What is MVC?
- What are frameworks?
- FW/1 Basics
- Dependency Injection
- FW/1 Handy Features

So to summarise:
- we talked about what frameworks are, why you'd use them, and a bit about what MVC is and how it applies to web frameworks
- we ran through the core features of FW/1 - views, layouts, controllers, and calling services
- and we talked about a few useful functions you use day-to-day in FW/1 like redirect, setView and populate

Thank you!

- https://github.com/seancorfield/fw1
  - FW/1 source, issues, pull requests
- “framework-one” on Google Groups - Mailing List
- https://github.com/seancorfield/di1
- https://github.com/arcins/fw1-intro-presentation
- https://github.com/marcins/cfobjective2012
- @MarcinS or marcins@webqem.com

webqem

ENDE

Thank you!

- https://github.com/seancorfield/fw1
    - FW/1 source, issues, pull requests
- "framework-one" on Google Groups - Mailing List
- https://github.com/seancorfield/di1
- https://github.com/arcins/fw1-intro-presentation
- https://github.com/marcins/cfobjective2012
- @MarcinS or marcins@webqem.com

webqem

Wednesday, 13 March 13

ENDE