Atomic Reactor!

Or how I learned to stop worrying and love the ORM

Mark Drew
Railo Technologies
CFUnited
2009

Who is Mark Drew?

- Railo UK's CEO
  - Fast Open Source CFML Engine
  - Development Consultancy
- CFEclipse's Lead Developer
- UK ColdFusion User Group's Co-Manager
- A CFML Developer since '97 and Web developer since '94 (showing my age now eh?)
- Reactor ORM Project Manager! (hence I am here I guess!)

What is this about then?

- Looking at how we access the database so far
- Getting to know some Patterns
- Getting to know how to use Reactor

Accessing the database so far has been:

```
<cfquery name="getUsers" datasource="myDSN">
        SELECT *
        FROM Users
</cfquery>
<cfoutput query="getUsers">
        <p>#firstName# #lastName#</p>
</cfoutput>
```

Why is this a problem?

- Because for each table in our systems we have to write multiple queries, usually:
  - Create: Insert a new row
  - Read: Select one or more rows
  - Update: Save values to that row
  - Delete: Delete an row from a table

What is this called?

(Not really a nice name is it?)

Why is this a problem?

- Time consuming, imagine we have 3 tables:

minutes/ action	Create	Read	Update	Delete	**TOTAL**
Post	10	2	10	2	
Comment	8	2	8	2	
User	6	3	7	2	
TOTAL					

Why is this a problem?

- Time consuming, imagine we have 3 tables:

minutes/ action	Create	Read	Update	Delete	**TOTAL**
Post	10	2	10	2	24
Comment	8	2	8	2	20
User	6	3	7	2	18
TOTAL	24	7	25	6	62

- So how long would it take for 20 tables?

6 hours

&

50 minutes!!!

Mark Drew - Railo

Why is this a problem?

- BOREDOM! Do you really want to code those actions again and again? I get bored easily!
- "We are going Open Source! We need it to run on MySQL now!" (not that it happens a lot, but you never know!)
- Repetitive tasks that can be error prone
- Where do you put this code? In each view?
- We added another column! You did remember to update the Create, Update and Read queries didn't you?

Design Patterns to the Rescue!

- DAO's
- Gateways
- Active Record

- Data Access Objects (DAOs)
  - Provide methods for Creating, Reading, Updating and Deleting a single record.
  - Generically called CRUD methods.
- Table Data Gateway Objects (Gateways)
  - Performs actions on multiple rows in the database.
  - Typically gateway methods return record sets.
- Active Records
  - Newer.
  - One object represents one record.
  - "Knows" how to read, write and itself from the database.
  - Provides getters/setters for modifying data

Design Patterns?

- Whoa? What are these design patterns?

Talking in tongues?

"Hey Bob, I wrote an app that needed to work on two database systems, and came up with a cool way of doing it!"

"Oh yeah? What's that, Jim?"

"I wrote a bunch of POJOs to represent my data, then wrote a bunch of database-specific objects to save and load my POJOs."

Mark Drew - Railo

The birth of a pattern

"Oh, I've done that too. I call it Bob's Super Hot Implicit Thingamajig."

"I'm not sure that's a good name to present to management. Let's call it *Data Access Object* from now on."

"Oh, ok. That's not as much fun, but now we can just say 'I wrote a DAO app!' and we both know the whole idea!"

Mark Drew - Railo

Some Drawbacks to the New Way

- It's time consuming
  - Writing a one-off query only takes a minute
  - Writing a series of CFCs (and testing them) takes a lot longer.
- It's verbose
  - A typical query is only a few lines of code.
  - Adding CFCs into the mix adds a lot of extra CFML
- It's repetitive
  - Most CFCs end up looking almost identical.
  - Only file, table and column names typically change.
  - Leads to a copy-paste-and-edit mentality.
  - Leads to subtle bugs, especially in rarely used code.

Database Abstraction Generators to the Rescue!

- Many developers end up writing programs to generate this repetitive code.
  - May automatically inspect the database (or not)
  - Tend to create static files which are manually updated as needed.
- This technique has it's own problems:
  - What if you customize an object but later add a new field to your database?

Enter Reactor…

- Reactor is coined as an “Inline Dynamic Database Abstraction" API.
- The Reactor API is used in your code.
- Generates objects only as needed (and configured).
- Instantiates objects and returns them for you.

A Simple Example:

```
<cfset reactor = CreateObject("Component",
                  "reactor.reactorFactory")
                 .init(expandPath("reactor.xml")) />
```

```
<cfset Address = reactor.createRecord("Address") />

<cfset Address.load(id=1234) />

<cfset Address.setStreet("My Street")>

<cfset Address.save()>
```

What Reactor Is …

- A framework used to generate CFML objects to access data in your database
- Reactor automates much of the repetitive, tedious and error-prone work involved in creating an Object Oriented database abstraction layer.

What Reactor Isn't …

- Reactor is Not Ruby (or CFML) on Rails
  - Does not generate application controllers
  - No Scaffolding
  - Relies on XML, not conventions
- Reactor is Not a Panacea*
  - It does not do everything you need it to!
  - You will need to customize some reactor generated objects.

* solves everything

Supported DBMS

- Microsoft SQL Server 2000 and 2005
- MySQL 4
- MySQL 5 and later
- PostgreSQL
- Oracle 9i and 10g
- DB2

Getting Reactor

- Subversion
  - http://svn.reactorframework.com/reactor/trunk/
- RC 1
  - http://trac.reactorframework.com
- Useful folders:
  - /Reactor
  - /ReactorSamples
  - /Documentation

Version 1??

- There are a few things to get it to V 1.0
- Mainly website, documentation.
- Very stable code. (apart from bugs with lesser used databases)
- Version 2.0 coming next year with some exciting additions (more on that later.... maybe, if you behave)

Installing Reactor

- Place "/reactor" in your web root.
- Or…
- Make a mapping "/reactor" to the Reactor folder.
- That's it!

Configuring Reactor

- Configuration in Reactor.xml

```
<reactor>
  <config>
    <type value="mysql" />
    <project value="CFCamp"/>
    <dsn value="CFCamp" />
    <mapping value="/model"/>
    <mode value="development"/>
  </config>

  <objects />
</reactor>
```

Create the ReactorFactory

```
<cfset application.reactor =
        CreateObject("component", "reactor.reactorFactory")
        .init("Reactor.xml")>
```

Create Records and Gateways

```
<!--- Create a post record --->
<cfset post =
  application.reactor.createRecord("post")>
<cfdump var="#post#">

<!--- Create a gateway to the post table --->
<cfset postGateway =
application.reactor.createGateway("post")>
<cfdump var="#postGateway#">
```

What happened there?

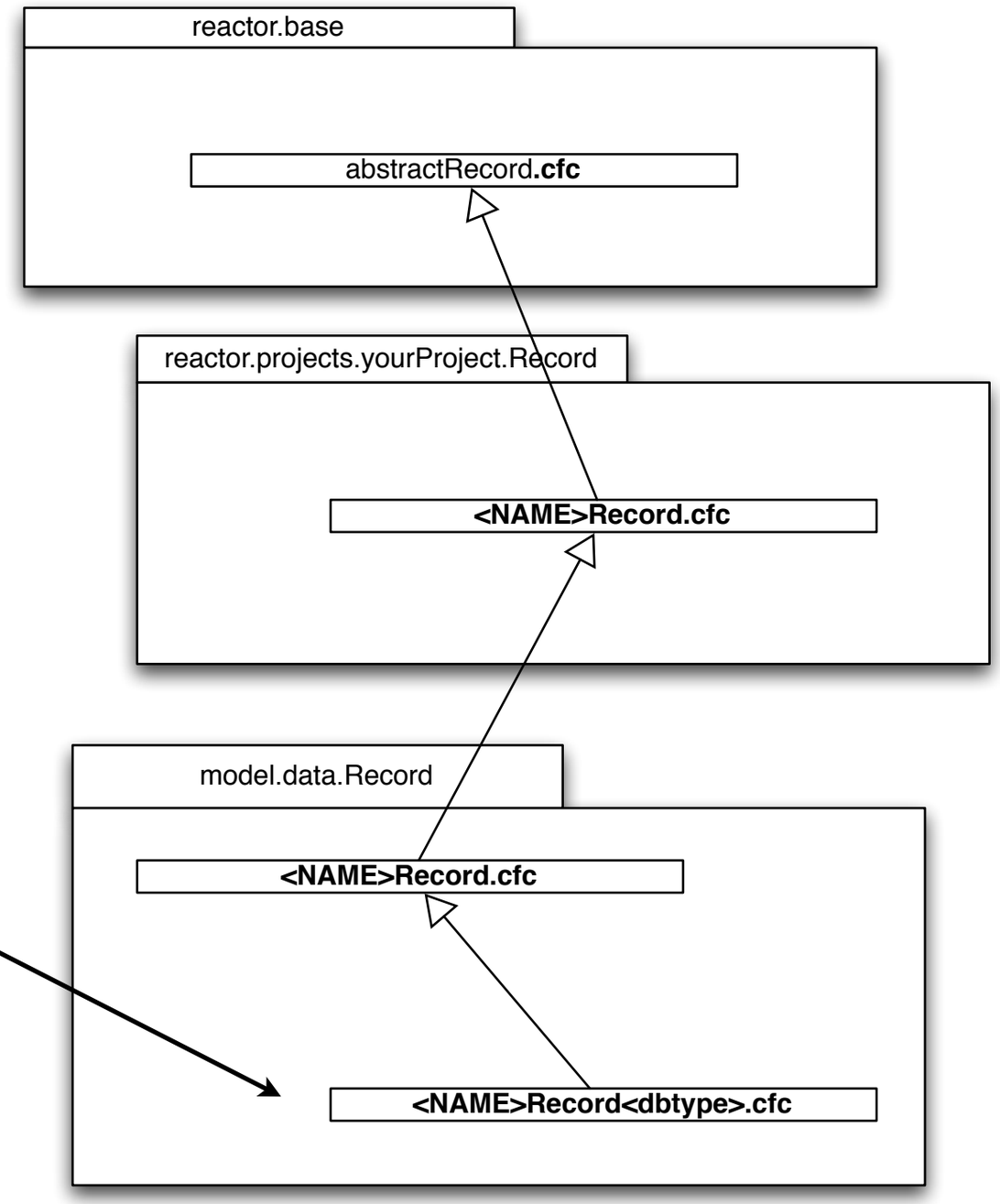
- Under *reactor/project/<our project>/* Reactor created our Base CFC's for us as well as any queries we required
- Under our model, Reactor created our editable Records and Gateways

Editable and extendable

- Reactor created two types files in our model folder for us to edit:
  - <Table>Record.cfc : For all databases
  - <Table>Recordmysql.cfc : MySQL Specific
- This goes for Gateways too
- <Table>Record.cfc to do generic functions or override (such as getFullName instead of getFirstName & getSurname) NB: Decorator Pattern!
- <Table>Recordmysql.cfc to write queries that are specific to MySQL

Le Process...

```
reactor.createRecord(<NAME>)
```



Mark Drew - Railo

Defining your objects

- So far, Reactor has done all the work for us.

- But sometimes we need to tell Reactor about our objects, or call them something else (without modifying the database!)

Defining your Objects

```
<objects>
  <object name="post" />
  <object name="comment" />
</objects>
```

But we might want them to have another name, an alias:

```
<objects>
    <object name="post" alias="BlogPost"/>
    <object name="comment" alias="BlogComment"/>
</objects>
```

Relationships

- ... lets talk about them.
- No, I am not going to be your Dr of Love. (you have to fix that yourself!)!
- Tables relate to each other, so should objects.
- There are two types of relationships:
  - hasOne
  - hasMany
- Reactor provides an easy way to relate our objects...

Relationships

```
<object name="post">
    <hasMany name="comment">
        <relate from="id" to="postid"/>
    </hasMany>
</object>

    <object name="comment">
        <hasOne name="post">
            <relate from="postid" to="id"/
>
        </hasOne>
    </object>
```

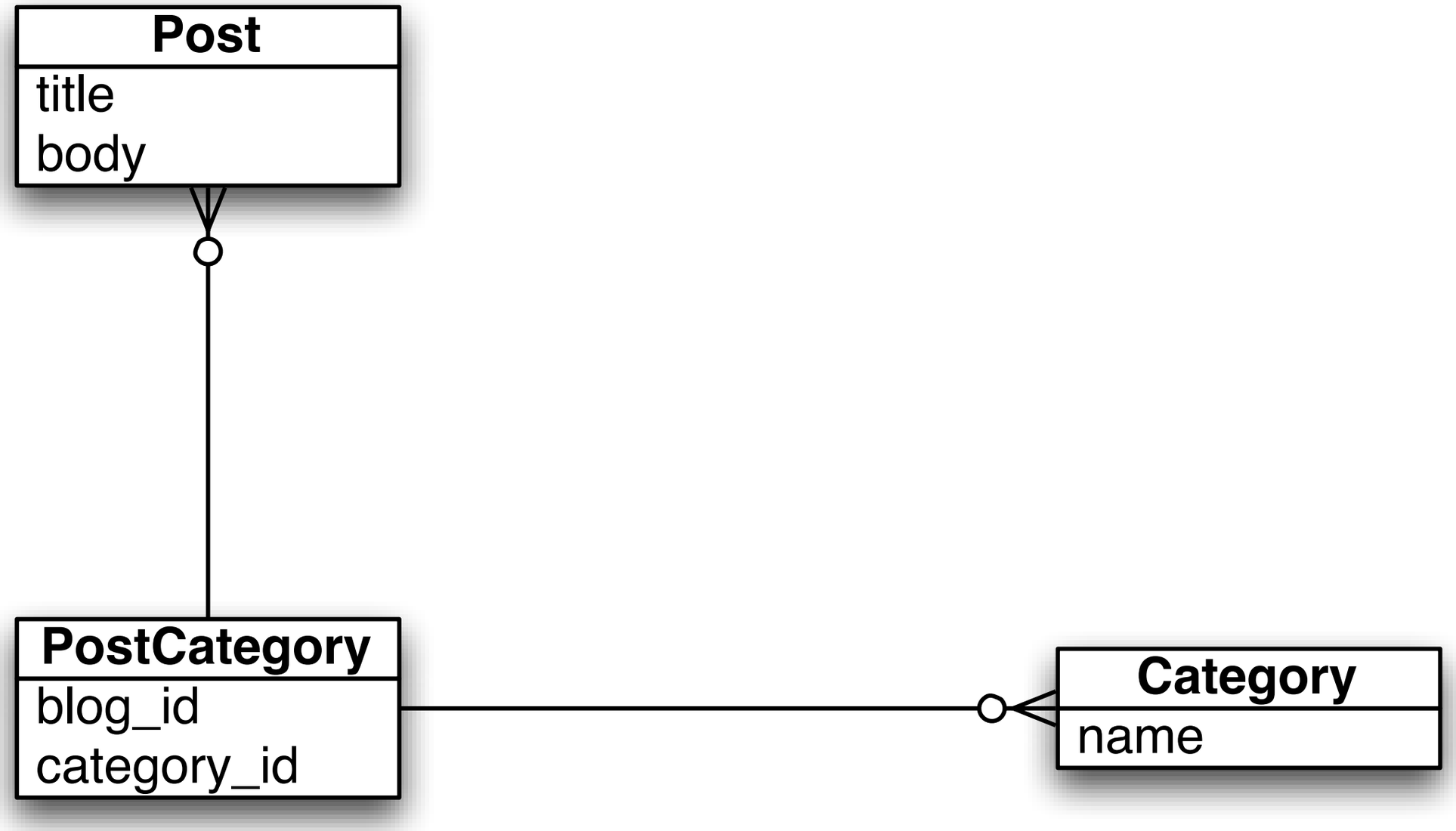
Relationships

- hasOne:
  - <Object>.get<RelatedObject>
  - <Object>.remove<RelatedObject>
  - <Object>.set<Related Object>

Relationships

- hasMany:
  - <Object>.get<RelateObject>Iterator

Some relationships are hard



Mark Drew - Railo

Linking Tables

- We define the linking table:

```
<object name="postcategory">
    <hasOne name="post">
        <relate from="postid" to="id" />
    </hasOne>
    <hasOne name="category">
        <relate from="categoryid" to="id" />
    </hasOne>
</object>
```

- then we define the link:

```
<object name="post">
    ...
    <hasMany name="category">
        <link name="postcategory" />
    </hasMany>
</object>
```

Linking Tables

- We can now just access it like another hasMany relationship:
- <Object>.get<Related Object>Iterator

Finally...

- In the same time it would have taken me to create all the queries for 3 tables...
  - I have also related them
  - Know that my commits are transaction safe
  - Done this presentation
  - Did a calculation how long it would take to create 20 tables
  - Shown you Reactor at work

Finally...

- There is more you can do with Reactor:
  - Object Oriented Queries
  - Add your own queries
- The project is starting up again and new features you can look forward to are:
  - Amazon Simple DB integration/support
  - Hibernate support (when Railo/Centaur support it)
  - Soft Delete, Ignore columns, IBO (Thanks to Peter Bell and Tom Chiverton)
  - Creation of Tables from definitions (quick development anyone?)
  - Validation framework (validate extend things such as emails etc)
  - **getByFilter (wanna see??)**
  - **beanInjectors! (get em while they are hot!)**

Mark Drew - Railo

getByFilter

- Gets filtered results as a recordset or iterator
- Easy to do complex filtered queries

getByFilter

```
Filtered = PostGateway.getByFilter(
        include={approved=1},
        exclude={deleted=1},
        contains={content="elvis"},
        orderby={id="ASC"},
        format="query",
        page=1,
        rows=10
        );
```

Injectors

- You can configure Reactor through ColdSpring
- You can add an “Injector” that adds beans from ColdSpring to any object created by Reactor

Injectors

```
   <!-- the service we want to inject -->
<bean id="TestService" class="TestService" />
```

```
<!-- Create an injector for our gateway -->
<bean id="PostRecordInjector" class="reactor.core.Injector">
   <property name="target"><value>PostGatewayMySQL</value></property>
   <property name="beans"><value>TestService</value></property>
</bean>
```

Injectors

You can now access the injected bean within your Reactor objects like:

```
<cffunction name="doTest">
    <cfreturn variables.TestService.doTest()>

</cffunction>
```

Resources

- Documentation
  - A work in progress (with framework) Ezra?!
- Trac Site
  - http://trac.reactorframework.com
    - Roadmap, timeline, tickets, and wiki
- http://www.alagad.com/go/blog
- http://www.markdrew.co.uk/
- Reactor Mailing List (s) on google groups:
  - http://groups.google.com/group/reactor-users
  - http://groups.google.com/group/reactor-devel

Questions / Answers