End-to-End Application Design

Featuring ColdSpring/AOP, FW/1,
MXUnit, ORM and ValidateThis

Jamie Krug
jamiekrug.com/blog
identi.ca/jamiekrug
twitter.com/jamiekrug
github.com/jamiekrug

cf.Objective() 2011

Goals

- Good application design
- Progressive development, refactoring
- Frameworks, patterns, tools
- Code with confidence (down with design paralysis!)

Application Design: "10,000-foot view"

- Model-View-Controller (MVC)
- Dependency Injection (DI)
- Object-Relational Mapping (ORM)
- Aspect-Oriented Programming (AOP)
- Unit Testing, Test-Driven Development (TDD)

Sample Application Overview

- MVC with Framework One (FW/1)
- Data Access Layer of Model
- Service Layer of Model
- Persistence with CFML/Hibernate ORM
- ValidateThis for CFC Validation
- ColdSpring Dependency Injection (DI)
- Abstract Gateway/Service for ORM Entities
- ColdSpring AOP for Transaction Advice
- Abstract Persistent Entity CFC
- Unit Testing with MXUnit

DEMO
Sample Application...

Model-View-Controller (MVC)

A pattern to isolate business/domain logic from the user interface.

Model: domain objects, business logic, data access

View: user interface (input and presentation)

Controller: coordination between model and view

MVC with Framework One (FW/1)

- Simple, lightweight, flexible, powerful MVC
- Convention over configuration (no XML, no "boilerplate")
- The "invisible" framework (no API restrictions/requirements)
- Single framework.cfc (extended by Application.cfc)
- Supports any bean factory (ColdSpring, Lightwire, etc.)
- Auto-wiring of controllers with service dependencies
- Flexible options for wrapping views and layouts
- Custom "skinning" (dynamic view/layout selection)
- Subsystems (stand-alone or integrated sub-apps)
- "Search engine safe" (SES) URLs

FW/1 Convention: Request Action

/index.cfm?**action=*section.item***

Configurable default section ("main") and item ("default"):

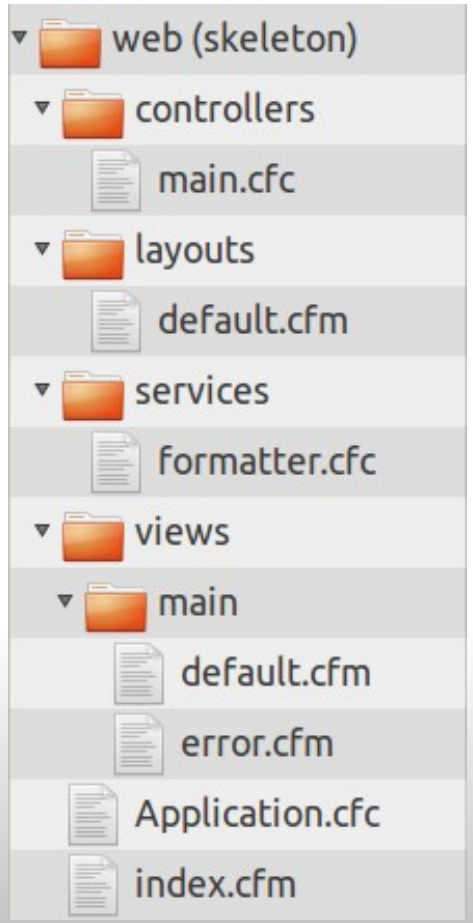
```
/index.cfm?action=main.default
/index.cfm?action=main
/index.cfm
```

Examples (basic and SES style):

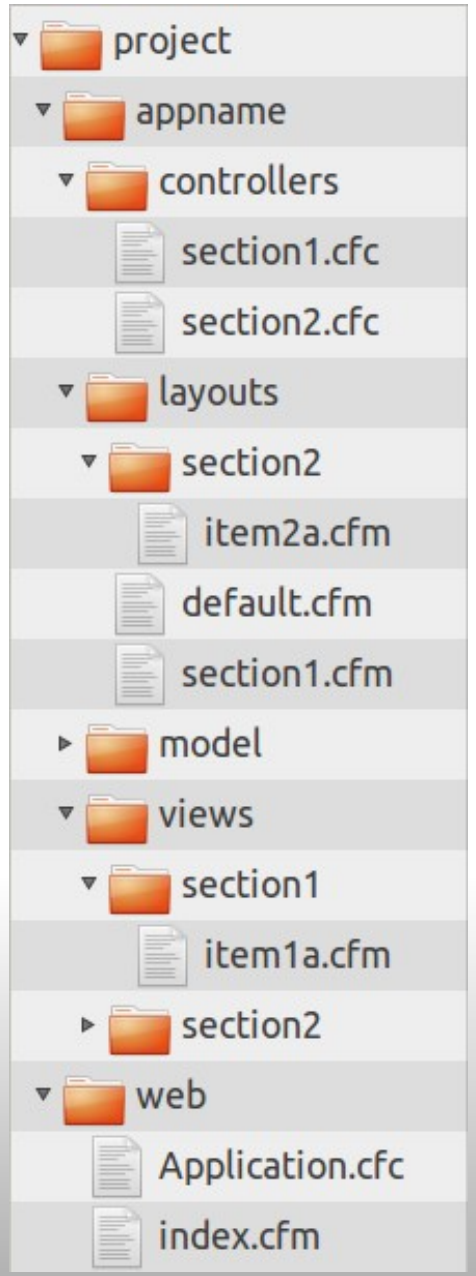
```
/index.cfm?action=user.list
/index.cfm/user/list

/index.cfm?action=user.edit&id=123
/index.cfm/user/edit/id/123

/index.cfm?action=security.logout
/index.cfm/security/logout
```

FW/1 Convention: File Organization

Default base
(*under* Web root)



Custom base
(*outside* Web root)
`base = '/appname'`



FW/1 Convention: Request Context

request.context struct contains merged URL/form values

rc is shorthand for the request context, available to:

- controllers (**rc** is sole argument to controller methods)
- views
- layouts

Controller methods can set data in **rc**, for views/layouts

FW/1 Convention: Request Flow-1/2

1. /controllers/*section*.cfc:*item*(rc)
2. /views/*section*/*item*.cfm -- **required**
3. /layouts/*section*/*item*.cfm
4. /layouts/*section*.cfm
5. /layouts/default.cfm

FW/1 Convention: Request Flow-2/2

1. /controllers/section.cfc:before(rc)
2. /services/section.cfc:item(argumentcollection = rc)
3. /controllers/section.cfc:after(rc)
4. /views/section/item.cfm -- **required**
5. /layouts/section/item.cfm
6. /layouts/section.cfm
7. /layouts/default.cfm

FW/1 Configuration, Application.cfc

```
variables.framework = {
    action = 'action',
    defaultSection = 'main',
    defaultItem = 'default',
    error = 'main.error',
    reload = 'reload',
    password = 'true',
    reloadApplicationOnEveryRequest = false,
    generateSES = false,
    SESOmitIndex = false,
    base = '/',
    baseURL = 'useCgiScriptName',
    suppressImplicitService = false,
    unhandledExtensions = 'cfc',
    unhandledPaths = '/flex2gateway',
    preserveKeyURLKey = 'fw1pk',
    maxNumContextsPreserved = 10,
    cacheFileExists = false,
    applicationKey = 'org.corfield.framework'
};
```

DEMO
FW/1 basics...

Model

- **Domain objects**
  - Most business logic and data
- **Data access layer**
  - All database/persistence operations
- **Service layer**
  - Main API or "gate keeper" to the model

Model: Domain/Business Objects

- Core business logic and data
- "Smart" objects, well-encapsulated
- Related objects, still loosely coupled
- Most important part of application model?

Model: Data Access Layer

- Database operations, no business logic
- Easily mocked for testing/prototyping
- Data operations involving related domain objects
- Avoid DAO/Gateway-per-domain-object pattern

Model: Service Layer

- Keep service objects "thin"
  (and domain objects "smart")
- Operations involving multiple domain objects
- Data access layer interactions
- Avoid service-per-domain-object pattern

Persistence: CFML/Hibernate ORM

- Object-relational mapping
- Design for model, not data schema
- Saves time and lines of code
- Flexible, powerful

ORM Configuration

Configure ORM in Application.cfc:

```
this.name = 'orm_app';

this.datasource = 'mydsn';

this.ormEnabled = true;

this.ormSettings = {
    cfclocation = '/appname',
    dbcreate = 'update',
    flushatrequestend = false
};
```

ORM Entities

```
component persistent='true' hint='User.cfc'
{
   property name='id' fieldtype='id' generator='native';
   property name='dateCreated' ormtype='timestamp';
   property name='firstName';
   property name='lastName';
}
```

ORM CRUD

```
transaction {
    user = entityNew( 'User', { 'name' = 'Jamie' } );
    entitySave( user ); // Create
}

id = user.getID();

transaction {
    user = entityLoadByPK( id ); // Read
    user.setName( 'Jamie Krug' );
    entitySave( user ); // Update
}

transaction {
    user = entityLoadByPK( id );
    entityDelete( user ); // Delete
}
```

Importance of ORM Transactions

- Encapsulate units of work
- Demarcation controlled by business logic
- Explicit control of *when* database operations occur
- Avoid accidental database changes

DEMO
ORM, Service/Data Layers...

Questions?

Part 2, after a short break...

ValidateThis for CFC Validation

- Why a validation framework? DRY
- Encapsulate validation rules in one place
- Client and server side validation
- Single object, multiple contexts
- Built-in or custom validation types
- Failure messages, default or custom

ValidateThis, Quick Start

Validation service (singleton):

```
import ValidateThis.ValidateThis;
validateThisConfig = { jsRoot = '/js/', definitionPath = '/model/' };
application.validationService = new ValidateThis( validateThisConfig );
```

Server-side validation of an object:

```
user = new User( 'Jamie' );
result = application.validationService.validate( user );
if ( !result.getIsSuccess() )
    // handle validation failures
```

Generate client-side validations:

```
<!--- output between <head></head> tags --->
#application.validationService.getValidationScript( objectType = 'User' )#
```

Built-in validation types Javascript initialization (if needed):

```
<!--- output between <head></head> tags --->
#application.validationService.getInitializationScript()#
```

ValidateThis Rules Definition

```xml
<?xml version="1.0" encoding="UTF-8"?>
<validateThis
        xsi:noNamespaceSchemaLocation="/ValidateThis/core/validateThis.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <objectProperties>
        <property name="name" desc="Your Name">
            <rule type="required" failureMessage="Your name is required." />
        </property>
        <property name="email" desc="E-mail">
            <rule type="required" failureMessage="E-mail is required." />
            <rule type="email" failureMessage="E-mail must be a valid e-mail address." />
            <rule type="custom" failureMessage="E-mail is already in use by an existing user.">
                <param methodname="emailUniqueValidator" />
            </rule>
        </property>
    </objectProperties>
</validateThis>
```

DEMO
ValidateThis...

ColdSpring Framework

- Power of Java Spring framework, for CFML
- Dependency Injection (DI) framework
- Aspect-Oriented Programming (AOP) framework
- Ease configuration and dependencies of CFCs
- Remote proxies, automatically generated
- (Much more)

ColdSpring: Beans Definition

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans default-autowire="byName">

    <bean id="userService" class="coldspring.examples.quickstart.components.UserService" />

    <bean id="userGateway" autowire="no" class="coldspring.examples.quickstart.components.UserGateway">
        <property name="configBean">
            <ref bean="configBean" />
        </property>
    </bean>

    <bean id="configBean" class="coldspring.examples.quickstart.components.ConfigBean" />

</beans>
```

ColdSpring: Bean Factory

```
import coldspring.beans.DefaultXmlBeanFactory;

coldspringConfig = '/appname/config/coldspring.xml';
beanFactory = new DefaultXmlBeanFactory();
beanFactory.loadBeans( coldspringConfig );

userService = getBean( 'userService' );

users = userService.listUserByFirstName( 'Jamie' );
```

DEMO
ColdSpring DI...

Abstract Gateway/Service for ORM

- Encapsulate generic ORM CRUD
- Dynamic and flexible with onMissingMethod():
  - getXXX(id)
  - getXXXByYYY(yyyValue)
  - listXXX[FilterByYYY][OrderByZZZ]([yyyValue])
  - newXXX()
  - saveXXX(object)
  - deleteXXX(object)
- Extend as needed

DEMO
Abstract Gateway/Service...

ColdSpring AOP for Transaction Advice

- Transactions transparent to business logic
- DRY (cross-cutting concerns)
- Simplify service methods
- Wrap multiple service methods

DEMO
AOP Transaction Advice...

Abstract Base CFCs

- DRY, e.g.,
  - Entity properties (id, created, lastModified...)
  - Common service dependencies (DI)
  - Common/generic behavior
- Rich, shared behavior and data
- Free to override methods, if needed
- E.g., AbstractPersistentEntity.cfc as mappedsuperclass

DEMO
Abstract Persistent Entity...

Unit Testing with MXUnit

- What is unit testing?
- When do I perform unit testing?
- Where do I perform unit testing?
- How do I perform unit testing?

DEMO
MXUnit unit testing...

Review: Big Picture

- Controllers -- KISS
  - Framework/HTTP coordination
  - A good service layer API helps!
- Service layer -- KISS
  - How you expose your model API
  - API useful to controllers, remote/Web Service
- Domain objects
  - "Smart" objects encapsulating domain logic
  - Keep OOP best practices in mind when designing
- Abstract CFCs
  - DRY--but don't stress either--refactoring later is OK
- Unit testing
  - Use it and enjoy the safety net!

Review: Down w/Design Paralysis!

There are *good* ways to "hack something together quickly."

The key is to know how/where to allow for refactoring.

- "Heavy" controllers and/or services
- Data access in controllers and/or services
- Mocked data access layer
- Avoid premature performance tuning

Questions?

Thank you!

Jamie Krug
jamiekrug.com/blog
identi.ca/jamiekrug
twitter.com/jamiekrug
github.com/jamiekrug

End-to-End Application Design

Featuring ColdSpring/AOP, FW/1,
MXUnit, ORM and ValidateThis

Jamie Krug
jamiekrug.com/blog
identi.ca/jamiekrug
twitter.com/jamiekrug
github.com/jamiekrug

cf.Objective() 2011

Goals

- Good application design
- Progressive development, refactoring
- Frameworks, patterns, tools
- Code with confidence (down with design paralysis!)

Tenets of good application design in CFML context.

Knowing how to "plan" for refactoring.

Frameworks/patterns as *tools* _not_ *obstacles*.

Learning to be "okay" with moving on w/development.

Learning new stuff can be difficult--this session strives to cover things that can provide a big ROI of learning time.

Application Design: "10,000-foot view"

- Model-View-Controller (MVC)
- Dependency Injection (DI)
- Object-Relational Mapping (ORM)
- Aspect-Oriented Programming (AOP)
- Unit Testing, Test-Driven Development (TDD)

MVC:
architectural pattern to isolate business logic from user interface

DI:
allows a software component to list dependencies, and a DI framework handles the dependency resolution (DI is a specific form of the IoC principle)

ORM:
abstracts persistence of objects (CFC instance data) to relational database

AOP:
increase modularity and decrease coupling by allowing for separation of cross-cutting concerns

Sample Application Overview

- MVC with Framework One (FW/1)
- Data Access Layer of Model
- Service Layer of Model
- Persistence with CFML/Hibernate ORM
- ValidateThis for CFC Validation
- ColdSpring Dependency Injection (DI)
- Abstract Gateway/Service for ORM Entities
- ColdSpring AOP for Transaction Advice
- Abstract Persistent Entity CFC
- Unit Testing with MXUnit

Expectations/Disclaimers:

* Variety of topics: wide, yet related
* Depth of topics: shallow individually, yet rich and powerful collectively
* Tools: excellent choices, but not the only choices
* Examples: directly useful, but not the only way
* At end we'll review how/when to "break the rules" and refactor later

DEMO
Sample Application...

Browser preview of branch:
presentation_cfObjective-2011

Model-View-Controller (MVC)

A pattern to isolate business/domain logic from the user interface.

Model: domain objects, business logic, data access

View: user interface (input and presentation)

Controller: coordination between model and view

MVC separation of concerns permitting independent development, testing and maintenance of each (Model/View).

MVC with Framework One (FW/1)

- Simple, lightweight, flexible, powerful MVC
- Convention over configuration (no XML, no "boilerplate")
- The "invisible" framework (no API restrictions/requirements)
- Single framework.cfc (extended by Application.cfc)
- Supports any bean factory (ColdSpring, Lightwire, etc.)
- Auto-wiring of controllers with service dependencies
- Flexible options for wrapping views and layouts
- Custom "skinning" (dynamic view/layout selection)
- Subsystems (stand-alone or integrated sub-apps)
- "Search engine safe" (SES) URLs

Heavy list for one slide, but good highlights to intro FW/1.

Briefly explain each point.

FW/1 Convention: Request Action

/index.cfm?**action=*section.item***

Configurable default section ("main") and item ("default"):

```
/index.cfm?action=main.default
/index.cfm?action=main
/index.cfm
```

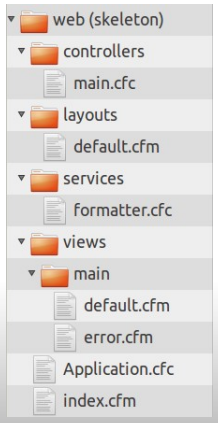
Examples (basic and SES style):

```
/index.cfm?action=user.list
/index.cfm/user/list

/index.cfm?action=user.edit&id=123
/index.cfm/user/edit/id/123

/index.cfm?action=security.logout
/index.cfm/security/logout
```

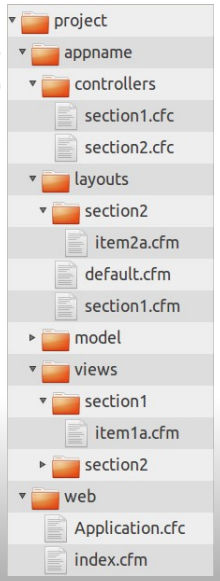
How to make a request of the framework--note nomenclature for later discussion:
action --> section and item

FW/1 Convention: File Organization

Default base
(*under* Web root)



Custom base
(*outside* Web root)
`base = '/appname'`



layouts == main display layout template(s)

views == body markup and other "pods" of display output

controllers == framework/request-"aware" CFCs

Review example screen shot directory structures, referencing action=section.item from prior slide.

FW/1 Convention: Request Context

request.context struct contains merged URL/form values

rc is shorthand for the request context, available to:

- controllers (**rc** is sole argument to controller methods)
- views
- layouts

Controller methods can set data in **rc**, for views/layouts

RC is basically data *from* user/request and data *to* views/layouts/response.

FW/1 Convention: Request Flow-1/2

1./controllers/*section*.cfc:*item*(rc)
2./views/*section*/*item*.cfm -- **required**
3./layouts/*section*/*item*.cfm
4./layouts/*section*.cfm
5./layouts/default.cfm

FW/1 Convention: Request Flow-2/2

1. /controllers/section.cfc:before(rc)
2. /services/section.cfc:item(argumentcollection = rc)
3. /controllers/section.cfc:after(rc)
4. /views/section/item.cfm -- **required**
5. /layouts/section/item.cfm
6. /layouts/section.cfm
7. /layouts/default.cfm

Discuss implicit service calls by FW/1, and setting to turn off.

FW/1 Configuration, Application.cfc

```
variables.framework = {
    action = 'action',
    defaultSection = 'main',
    defaultItem = 'default',
    error = 'main.error',
    reload = 'reload',
    password = 'true',
    reloadApplicationOnEveryRequest = false,
    generateSES = false,
    SESOmitIndex = false,
    base = '/',
    baseURL = 'useCgiScriptName',
    suppressImplicitService = false,
    unhandledExtensions = 'cfc',
    unhandledPaths = '/flex2gateway',
    preserveKeyURLKey = 'fw1pk',
    maxNumContextsPreserved = 10,
    cacheFileExists = false,
    applicationKey = 'org.corfield.framework'
};
```

DEMO
FW/1 basics...

demo_01_FW1-basics

Walk through action=main.default:

* Application.cfc
* Browser: home page
* layouts/default.cfm
* views/main/default.cfm
* controllers/main.cfc

Model

- **Domain objects**
  - Most business logic and data
- **Data access layer**
  - All database/persistence operations
- **Service layer**
  - Main API or "gate keeper" to the model

Your focus should be on domain objects as most important part of OOP design.

Data access is just that.

Service layer basics--more covered later.

Model: Domain/Business Objects

- Core business logic and data
- "Smart" objects, well-encapsulated
- Related objects, still loosely coupled
- Most important part of application model?

Model: Data Access Layer

- Database operations, no business logic
- Easily mocked for testing/prototyping
- Data operations involving related domain objects
- Avoid DAO/Gateway-per-domain-object pattern

Model: Service Layer

- Keep service objects "thin"
  (and domain objects "smart")
- Operations involving multiple domain objects
- Data access layer interactions
- Avoid service-per-domain-object pattern

Persistence: CFML/Hibernate ORM

- Object-relational mapping
- Design for model, not data schema
- Saves time and lines of code
- Flexible, powerful

ORM can be overwhelming to the uninitiated, but it has big ROI potential!

Think about domain model objects and good OOP principles--not database.

ORM Configuration

Configure ORM in Application.cfc:

```
this.name = 'orm_app';

this.datasource = 'mydsn';

this.ormEnabled = true;

this.ormSettings = {
    cfclocation = '/appname',
    dbcreate = 'update',
    flushatrequestend = false
};
```

ORM Entities

```
component persistent='true' hint='User.cfc'
{
   property name='id' fieldtype='id' generator='native';
   property name='dateCreated' ormtype='timestamp';
   property name='firstName';
   property name='lastName';
}
```

ORM CRUD

```
transaction {
    user = entityNew( 'User', { 'name' = 'Jamie' } );
    entitySave( user ); // Create
}

id = user.getID();

transaction {
    user = entityLoadByPK( id ); // Read
    user.setName( 'Jamie Krug' );
    entitySave( user ); // Update
}

transaction {
    user = entityLoadByPK( id );
    entityDelete( user ); // Delete
}
```

Importance of ORM Transactions

- Encapsulate units of work
- Demarcation controlled by business logic
- Explicit control of *when* database operations occur
- Avoid accidental database changes

DEMO
ORM, Service/Data Layers...

demo_02_orm-service-data-basics

* Application.cfc
* Abbreviation.cfc
* Definition.cfc
* AbbreviationService.cfc
* AbbreviationGateway.cfc
* Browser: home page
* action=main.default: layout, view, controller, service, gateway
* action=abbreviation.define: view, controller, service, gateway
* action=abbreviation.submit: controller, service, gateway

Questions?

Part 2, after a short break...

Covered So Far:

* MVC
* Model: domain objects, data access, services
* FW/1
* ORM

ValidateThis for CFC Validation

- Why a validation framework? DRY
- Encapsulate validation rules in one place
- Client and server side validation
- Single object, multiple contexts
- Built-in or custom validation types
- Failure messages, default or custom

ValidateThis, Quick Start

Validation service (singleton):

```
import ValidateThis.ValidateThis;
validateThisConfig = { jsRoot = '/js/', definitionPath = '/model/' };
application.validationService = new ValidateThis( validateThisConfig );
```

Server-side validation of an object:

```
user = new User( 'Jamie' );
result = application.validationService.validate( user );
if ( !result.getIsSuccess() )
    // handle validation failures
```

Generate client-side validations:

```
<!--- output between <head></head> tags --->
#application.validationService.getValidationScript( objectType = 'User' )#
```

Built-in validation types Javascript initialization (if needed):

```
<!--- output between <head></head> tags --->
#application.validationService.getInitializationScript()#
```

ValidateThis Rules Definition

```xml
<?xml version="1.0" encoding="UTF-8"?>
<validateThis
        xsi:noNamespaceSchemaLocation="/ValidateThis/core/validateThis.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <objectProperties>
        <property name="name" desc="Your Name">
            <rule type="required" failureMessage="Your name is required." />
        </property>
        <property name="email" desc="E-mail">
            <rule type="required" failureMessage="E-mail is required." />
            <rule type="email" failureMessage="E-mail must be a valid e-mail address." />
            <rule type="custom" failureMessage="E-mail is already in use by an existing user.">
                <param methodname="emailUniqueValidator" />
            </rule>
        </property>
    </objectProperties>
</validateThis>
```

DEMO
ValidateThis...

demo_03_ValidateThis

* Browser: demo validation failures
* Application.cfc
* Abbreviation.xml
* Definition.xml
* AbbreviationService.cfc:
saveAbbreviation()/saveDefinition()

ColdSpring Framework

- Power of Java Spring framework, for CFML
- Dependency Injection (DI) framework
- Aspect-Oriented Programming (AOP) framework
- Ease configuration and dependencies of CFCs
- Remote proxies, automatically generated
- (Much more)

ColdSpring: Beans Definition

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans default-autowire="byName">

    <bean id="userService" class="coldspring.examples.quickstart.components.UserService" />

    <bean id="userGateway" autowire="no" class="coldspring.examples.quickstart.components.UserGateway">
        <property name="configBean">
            <ref bean="configBean" />
        </property>
    </bean>

    <bean id="configBean" class="coldspring.examples.quickstart.components.ConfigBean" />

</beans>
```

ColdSpring: Bean Factory

```
import coldspring.beans.DefaultXmlBeanFactory;

coldspringConfig = '/appname/config/coldspring.xml';
beanFactory = new DefaultXmlBeanFactory();
beanFactory.loadBeans( coldspringConfig );

userService = getBean( 'userService' );

users = userService.listUserByFirstName( 'Jamie' );
```

DEMO
ColdSpring DI...

demo_04_ColdSpringDI

* beans.xml
* AbbreviationService.cfc properties:
abbreviationGateway, validationService

Abstract Gateway/Service for ORM

- Encapsulate generic ORM CRUD
- Dynamic and flexible with onMissingMethod():
  - getXXX(id)
  - getXXXByYYY(yyyValue)
  - listXXX[FilterByYYY][OrderByZZZ]([yyyValue])
  - newXXX()
  - saveXXX(object)
  - deleteXXX(object)
- Extend as needed

DEMO
Abstract Gateway/Service...

demo_05_AbstractGateway

* AbstractGateway.cfc
* oMM tricks/"API"
* AbbreviationService.cfc

demo_06_AbstractService

* AbstractService.cfc: get/list methods all removed
* onMM "API"/passthrough
* Controllers -- calling missing service methods ;-)

ColdSpring AOP for Transaction Advice

- Transactions transparent to business logic
- DRY (cross-cutting concerns)
- Simplify service methods
- Wrap multiple service methods

DEMO
AOP Transaction Advice...

demo_07_AOPTransactionAdvice

* beans.xml
* TransactionAdvice.cfc
* AbbreviationService.cfc: transactions removed

Abstract Base CFCs

- DRY, e.g.,
  - Entity properties (id, created, lastModified...)
  - Common service dependencies (DI)
  - Common/generic behavior
- Rich, shared behavior and data
- Free to override methods, if needed
- E.g., AbstractPersistentEntity.cfc as mappedsuperclass

DEMO
Abstract Persistent Entity...

demo_08_AbstractPersistentEntity

* AbstractPersistentEntity and slimmed down beans

Unit Testing with MXUnit

- What is unit testing?
- When do I perform unit testing?
- Where do I perform unit testing?
- How do I perform unit testing?

DEMO
MXUnit unit testing...

demo_09_MXUnitTesting

* Review /tests directory structure
* Browser: /tests/unit/index.cfm -- show test suite run
* AbstractTestCase.cfc
* Review each test case CFC--*briefly*

Review: Big Picture

- Controllers -- KISS
  - Framework/HTTP coordination
  - A good service layer API helps!
- Service layer -- KISS
  - How you expose your model API
  - API useful to controllers, remote/Web Service
- Domain objects
  - "Smart" objects encapsulating domain logic
  - Keep OOP best practices in mind when designing
- Abstract CFCs
  - DRY--but don't stress either--refactoring later is OK
- Unit testing
  - Use it and enjoy the safety net!

Review: Down w/Design Paralysis!

There are *good* ways to "hack something together quickly."

The key is to know how/where to allow for refactoring.

- "Heavy" controllers and/or services
- Data access in controllers and/or services
- Mocked data access layer
- Avoid premature performance tuning

Questions?

Thank you!

Jamie Krug
jamiekrug.com/blog
identi.ca/jamiekrug
twitter.com/jamiekrug
github.com/jamiekrug

* Code on GitHub: https://github.com/jamiekrug/notintothewholebrevitything/tree/presentation_cfObjective-2011
* Hallway/lunch questions or discussion
* Feedback survey (4 questions) on cfObjective.com session page