"Building Sustainable ColdFusion Applications"

The Definitive Guide To The ColdBox Platform

(Covers up to version 2.6.3: Renewed)

By Luis F. Majano

ISBN 1449907865 EAN-13 9781449907860

First Edition

Luis F. Majano
ColdBox Platform
info@coldbox.org
www.coldbox.org

"But they that wait upon the LORD shall renew their strength; they shall mount up with wings as eagles; they shall run, and not be weary; and they shall walk, and not faint." Isaiah 40:31

To my beloved wife Veronica, te amo bbita!

Table of Contents

Forward 1

Preface 1

- Online Documents 1
- Audience For This Book 2
- How to Use This Book 2
- Book Overview 2
- License 3
- Online Resources 3
- Donations – Support Development 4
- How to Contact Us 4
- ColdBox Credits 5
- About the Author 7
- Charity Donations 7
- Personal Acknowledgments 8
- About the Technical Reviewer 9

Chapter 1 » Getting Started With ColdBox 11

- What is ColdBox? 11
- What Are Some of ColdBox's Novel Features? 13
- How ColdBox Works 16
- Implicit & Explicit Invocations 17
- Configuration File (coldbox.xml) 18
- Event Handlers (Controllers) 18
- Request Collection 19
- Plugins 19
- Interceptors 20
- Summary 22

Chapter 2 » Installing ColdBox 23

ColdBox Requirements 23
Optional Requirements 23
Typical Installation 23
Alternate Installation Methods 24
Upgrading ColdBox 25
Refactoring ColdBox 26
ColdBox Eclipse Plugins 31
ColdBox Syntax Dictionaries 33
Known Issues 35
Summary 35
Chapter 3 » ColdFusion Components & OO Terms 37
What is an Object? 37
What is a CFC? 38
Object Oriented Terms 38
ColdFusion Best Practices 40
Summary 46
Chapter 4 » Effective Web Application Architecture 47
What is MVC? 47
MVC Layers 48
Benefits of MVC 49
Domain Model 50
Service Layers 52
Gateways or Data Access Objects 53
Summary 53
Chapter 5 » ColdBox Essentials 55
ColdBox Request Life Cycles 55
Directory Structure & Conventions 58
Modifying Application Conventions 60
Modifying Framework-Wide Conventions 61
Implicit Execution Conventions 61

The Framework SuperType 64
Reserved Words & Methods 64
Event Handler Reserved Words & Methods 67
Plugin Reserved Words & Methods 67
Interceptor Reserved Words & Methods 68
ColdBox's URL Actions 69
The ColdBox SideBar 72
Summary 77
Chapter 6 » Internal Settings & Structures 79
The Internal Structures 79
Configuration Settings (coldbox.xml) 79
Main Setting Methods 81
How Do I Use Settings In My Model? 81
Configuration Structure and IoC Frameworks Like ColdSpring/LightWire? 82
Framework Settings Structure (Framework Settings) 82
Modifying Framework Settings 85
Summary 88
Chapter 7 » ColdBox Configuration File 89
Generated Schema Documentation 90
Loading Another Configuration File 90
Interacting with the Loaded Settings 90
${setting} Replacement 91
<Settings> Element 92
<YourSettings> 105
<Conventions> 106
<DebuggerSettings> 106
<MailServerSettings> 107
<BugTracerReports> 107
<WebServices> 108

<Layouts> 108
<i18N> 109
<Datasources> 111
<Cache> 111
<Interceptors> 112
Summary 113
Chapter 8 » The ColdBox Request Context 115
How Does It Work? 116
The Event Object 116
Getting a Reference To the Collection 116
What Can I Do With It? 116
Most Commonly Used Methods 117
Current request Metadata Methods 119
Extending the Request Context 119
For What This Be Used For? 120
How Does It Work? 120
Controller Calling 122
How to declare it 123
Summary 124
Chapter 9 » Event Handlers 125
What are Event Handlers? 125
How Are Events Called? 125
Event Handlers Location 127
Event Handlers External Location 127
Implicitly Declared Events 127
Rules and Anatomy of an Event Handler 128
Sample Handler Component Declaration 129
The Caching Parameters 129
Reserved Words and Methods 130

Sample Init() Method 131
Anatomy of an Event Handler Method 131
Event Caching 132
Method Samples 133
How To Set and Get Values (Event Handlers, Views, Layouts) 134
Relocating To Another Event 136
Setting Views 138
What If I Don't Want To Render Anything? 139
Rendering Data 139
Default Event Action 139
Event Handler Interceptors 140
onMissingAction() : Leveraging the Dynamic 142
Handler Public Property: this.EVENT_CACHE_SUFFIX 143
Persisting Flash Variables: ColdBox Flash RAM 143
Executing Events 145
Autowiring Your Handlers with Dependencies 146
Testing Handlers 146
Best Practices 147
Advanced OO Features: UDF Injections 148
Summary 148
Chapter 10 » SES URL Mappings 149
Benefits 149
Requirements 150
Configuring Your Application for SES Support 151
Loose Matching Property 152
The Routes Configuration File 153
Embedding Variables in a Route 157
Adding Variables Per Route 157
Numeric Routes 158

Optional Variables....159
Convention Name-Value Pairs....159
Route Examples....160
The Default Routes....160
Package Resolver....160
How Do You Relocate?....161
Automating Writing Links: event.buildLink()....163
How About Links On My Pages?....164
How About Form Submissions (Posts)?....164
The BASE Tag....164
Summary....165
Chapter 11 » Layouts & Views....167
What is a Layout?....167
What are Views?....170
Rendering Data....171
Implicit Declarations....172
Setting Views for Rendering....174
Rendering Without Layout....175
Implicit Views....175
Implicit View Helpers....176
Rendering & Caching Views....176
Purging Views....176
Overriding Layouts....177
Where Are Views/Layouts Rendered?....178
Methods/Properties You Can Use....178
Rendering Multiple Views....179
Content Variable Views....179
The ViewsExternalLocation Setting....181
Rendering External Views....181

Helper UDF's 181
Viewlets 182
Tips and Tricks 184
Summary 185
Chapter 12 » Working With Ajax 187
Returning HTML 187
Ajax HTML Layout 189
Returning DATA (XML/JSON/WDDX/Any) 190
The ColdBox Proxy 191
ColdBox Debugger 197
Summary 198
Chapter 13 » Internationalization (i18n) 199
Basics 199
Credits 199
coldbox.xml Declarations 199
Coding for i18n 200
Chaning Locales 205
Best Practices 205
Resources 206
Summary 206
Chapter 14 » Model Integration Guide 207
Model Layer Overview 208
Conventions Location 213
Models External Location 214
Model Configuration Options 214
Usage Methods 215
Dependencies DSL 216
Model Mappings 219
Persisting Model Objects 220
Simple Example 221

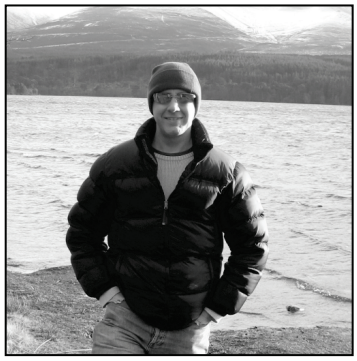
Summary 227
Chapter 15 » Plugins 229
What Are Plugins? 229
Core Plugins 229
Using Plugins 231
Creating Custom Plugins 233
Plugin Reserved Words & Methods 236
Autowiring Plugins 236
Advanced OO Features: UDF Injections 236
Best Practices 237
Summary 238
Chapter 16 » Interceptors 239
What Can They Be Used For? 240
How Interceptors Work 241
Order of Creation & Dependencies 246
Reserved Words & Methods 248
Core Interception Points 249
Core Interception Data 250
Interceptor Output Buffer 251
Custom Interception Points 251
Announcing Interceptions 253
Unregistering Interceptors 253
Appending Custom Interception Points 253
Register Interceptors at Runtime 254
Reporting Methods 255
Summary 255
Chapter 17 » Autowiring 257
What Are Dependencies? 257
Dependencies DSL 257
Configuring the Autowire Interceptor 262

Summary 263
Chapter 18 » The ColdBox Cache 265
ColdBox Cache Features 265
Inner Workings 266
Java Soft References 267
Configuring The Cache 268
Using The ColdBox Cache 270
Cache Monitor 271
Cache Panel Commands 272
Techniques 273
Plugin/Handler Caching 275
Interceptor Caching 276
View Caching 276
Event Caching 277
Summary 280
Chapter 19 » The ColdBox Proxy 281
Getting Started 283
The Base Proxy Object 283
Expanding the Proxy 284
The Configuration File 285
Execution Profiler Monitor 286
Remote Event Handlers 287
Distinguishing Request Types 287
Ajax Data Binding & More 288
Some Flex Code 289
Caveats & Gotchas 292
Summary 293
Chapter 20 » Integrating ColdBox 295
Environment Specific Configurations 295
Configuration 295

The Environments Configuration File 296
Extending the Interceptor 298
Deploy Interceptor 298
ColdSpring Integration 301
LightWire Integration 311
Transfer ORM Integration 314
Summary 323
Chapter 21 » Feed Reading & Generation 325
Feed Reading 325
Basic Usage 326
Configuring Your Application 327
Advance Usage 328
Feed Parsed Output 329
Feed Generator 333
Feed Generator Elements 347
Summary 354
Chapter 22 » Securing Your Applications 355
Features 355
How It Works? 356
Declaring the Interceptor 356
Rules 360
Sample XML Rules 362
_securedURL Key 364
Default Security 364
Custom Security: Validator Object 364
Summary 367
Chapter 23 » Unit Testing Handlers 369
What is Unit Testing? 369
How To Set It Up 370
Testing Handlers That Return Values 371

Asserting Relocations: setNextEvent and setNextRoute 372
The Base Test Case 372
Optional Setup: Application Start and Request Start Handler 373
ColdBox Factory and Application Scope 373
The Test Case: GeneralTest.cfc 374
Full Test Case 375
Important Caveats With Relative Paths 377
Important Caveats With FORM Elements 377
ColdBox Test Suites 379
Eclipse Integration With MXUnit 381
Summary 382
Appendix A » ColdBox Professional Services 383
Appendix B » License Agreement 385
Permitted Use for The ColdBox Framework 385
Use of ColdBox Logo, Web Assets, Documentation and Content 386
ColdBox Dashboard License 387
Index 389

Forward



There are a number of very capable frameworks for building ColdFusion/CFML applications. However ColdBox is both the most comprehensive and the most thoroughly documented. In addition to helping you to structure your application and making it easier to maintain, Coldbox also helps you to write applications that are easier to test, easier to internationalize and easier to scale.

In this book, Luis has brought together years of experience in developing high performance, maintainable web applications into a single, readable book designed to help people to do a better job of designing and building their applications.

All of the information you need to write great web applications exists online. But Luis has pulled together the key information - targeted at CFML developers using the ColdBox framework - to make it as quick and easy as possible to learn how to write better CFML applications using ColdBox.

If you are new to object oriented programming or frameworks, this book is an excellent investment. Starting with the basics of object oriented programming and best practices like DRY and MVC, it leads you through the process of understanding how to write maintainable applications using the ColdBox framework. If you're already a guru, you'll find lots of additional information on the latest features in ColdBox 2.6. Either way it makes a handy addition to your bookshelf for referring to when developing your applications.

At a time where there is still a lot of confusion about best practices for object oriented application development in CFML, hopefully this book will provide a starting point for developers interested in improving their skills - and their applications.

Peter Bell

New York, July, 2009

Preface

In late 2005 the first beta of ColdBox appeared on the public ColdFusion scene, even though the core framework and libraries where started back in 2003 out of a mission critical, high availability project. ColdBox was born with several key requirements: speed, stability, high availability, ease of use, and software aspects; Thus, becoming more than MVC. At that time, the business applications at my old employer needed to be much more than simple MVC methodologies; the plugin architecture of ColdBox was born from all the ideas of those projects and the fact that applications need so much more than just separation of concerns. This is where aspects like logging, bug reporting, caching, web services, and so much more started, from that need. However, I definitely saw that this need was shared among many developers in order to build small, medium or enterprise applications.

Due to the high success of the initial versions of the framework at that time, I decided to invest myself into developing a community initiative that became known as ColdBox. I saw the potential of conventions over configurations, in leveraging the dynamic nature of ColdFusion, in developing the framework as a ColdFusion application and not a Java application, and the idea of a development platform, not only a methodology. The idea started small but as usage grew and the potential for setting industry standards in a ColdFusion community where object orientation had just began, inspired me to dedicate myself wholeheartedly into the project. I was always taught that adoption comes at a price and that price is documentation. I was never a documentation guru, or even liked it, but it was something necessary in order to increase the adaptation rate of what I wanted to be an industry standard for developing ColdFusion applications. I had seen success first hand and wanted to share it to everybody I could.

Thus, my documentation obsession began. I tried to always comment as much as I could and force myself to add hints everywhere, because I knew documentation generation would be one of my best friends, and it has been ever since. I made it a mandate that no release, no matter how simple, would go out unless it was fully documented and ticketed, which I still maintain up to this day.

As you can see, the roots of ColdBox are ingrained in real life applications, and some even made companies millions of dollars a month due to its stability and progressiveness. ColdBox has quickly become one of the industry standards of enterprise ColdFusion development as we, Team Coldbox, are always striving to break the mold and innovate. With this inspiration at hand, you can now start delving into the ColdBox Platform and learn how you can start creating small, medium, or enterprise applications with ease.

Online Documents

The official ColdBox wiki contains the latest and greatest documentation and can be found at www.coldbox.org. At the time of this book writing, ColdBox 3.0.0 is in current development, so I encourage you to check the sites and resources for the latest ColdBox builds. Once ColdBox 3.0.0 is in the

wild, we will also update this book and create another release to support all of the features and enhancements ColdBox 3.0.0 will introduce. So enjoy this book and support ColdBox in any way you can.

Audience For This Book

This book assumes that the reader has at least basic ColdFusion knowledge and some understanding of MVC methodologies. This book does not go into detail about methodologies or provide as much instructional content as the *ColdBox Platform Training Series* courses do. If you are interested in professional training for you or your organization, please visit our training section at *www.coldbox.org/index.cfm/training* or send us an email at *training@coldbox.org*.

Who This Book Is For

This book is for developers or managers interested in having the latest ColdBox documentation in book format. This is an excellent reference book and companion to the online documentation. It will give you an insight of how the dynamic nature of ColdFusion can be taken to its limits by ColdBox, and provide you with an excellent RAD platform. So if you are already a ColdFusion or any dynamic language fanatic, this book will give you a jolt of electricity and an understanding of how ColdFusion can be leveraged to make your development faster, easier and more sustainable.

Furthermore, you will be propelled into learning new technologies and approaches to dynamic programming in ColdFusion that extends into the development of every layer of a typical web application. We do not hold back, we want to push the limits and be pioneers in the ColdFusion development arena. So if you feel like you need a challenge in your ColdFusion development and even Java development. Then this book will transform your outlook on ColdFusion/Java and Dynamic language development.

How to Use This Book

You can use this book as an excellent source of reference and introduction into the ColdBox Platform. This book is composed in a sequential order to give you the most from your reading. If you are looking for more instructional, step-by-step content, we encourage you to visit our training center: www.coldbox.org/index.cfm/training and find a suitable training course for you or your organization.

Book Overview

We begin with a high-rise view of the ColdBox Platform and start delving into the basics of object oriented ColdFusion, ColdFusion components and web application architecture. In proceeding chapters we cover the essentials of the ColdBox Platform and how to install and maintain it. By chapter 6 we will be immersed in how to configure ColdBox applications and will start seeing the power of this development platform. Then we move on to the major components of ColdBox, and discuss how to extend and integrate it into other frameworks and techonologies.

There is a full chapter on how to secure your ColdBox applications that will put a smile on your face, as you will be filled with joy of how easy it is to secure applications. We then finally conclude our journey with an in-depth look at Unit Testing, what we all developers LOVE to do; and I mean that!

License

The license of the ColdBox Platform is Apache License, Version 2.0:
http://www.apache.org/licenses/LICENSE-2.0

However, the contents of this book and all the online documentation, ColdBox logos and materials are exclusive property of **Ortus Solutions, Corp**. You cannot reproduce, distribute or sell this material without prior consent from **Ortus Solutions, Corp**. You can find more information about licensing in the Appendices.

Online Resources

Below are several resources that will help you in your ColdBox development and learning.

Official Website(s)

http://www.coldbox.org

http://www.luismajano.com

http://www.ortussolutions.com

Download ColdBox

http://www.coldbox.org/index.cfm/download

ColdBox Blog

http://blog.coldbox.org/

ColdBox Professional Support & Services

http://www.coldbox.org/index.cfm/support/overview

ColdBox Training

http://www.coldbox.org/index.cfm/training

ColdBox Google Mailing List

http://groups.google.com/group/coldbox

ColdBox Twitter Feed

http://twitter.com/coldbox

ColdBox Forums

http://forums.coldbox.org/

ColdBox Live API Docs

http://www.coldbox.org/api/

ColdBox Media Central

http://www.coldbox.org/index.cfm/media/tv

Railo CFML Engine

http://www.getrailo.com

Adobe ColdFusion

http://www.adobe.com

Open BlueDragon

http://www.openbluedragon.org

jQuery

http://www.jquery.com

MXunit

http://www.mxunit.com

Amazon Wishlist

http://www.amazon.com/wishlist/7DPYG3RZG3AF

Donations – Support Development

ColdBox is an open source initiative and it survives thanks to your donations. So please Donate to The ColdBox Framework or you can visit the Amazon Wishlist.

How to Contact Us

Please send us your comments, suggestions, and errata to info@coldbox.org with a subject line of **ColdBox Definitive Guide Errata**. We will promptly correct the errors or respond to your suggestions for our next release of this book.

ColdBox Credits

Team coldBox is always in search of people willing to install, test, and debug ColdBox to the max. Here you can see the people currently involved with ColdBox. They are implementing it on their sites, enhancing it, testing it or just playing around with it. We also give thanks and recognition to those open source projects and pieces of code that we have reused in ColdBox. If we have failed to mention code usage here, please let us know so we can add you.

Team ColdBox

- Luis Majano - http://www.luismajano.com
- Russ Johnson http://www.angry-fly.com
- Sana Ullah - http://www.sanaullah.co.uk/
- Rob Gonda - http://www.robgonda.com
- Matt Quackenbush - http://www.quackfuzed.com
- Tom de Manincor - http://www.tomdeman.com/
- Ernst Van Der Linden - http://evdlinden.behindthe.net

Contributors

- Brian LeGros - http://www.brianlegros.com
- Oscar Arevalo - http://www.oscararevalo.com
- Adam Fortuna for ColdCourse - http://www.adamfortuna.com
- Aaron Roberson
- Marc Esher - http://mxunit.org/blog/
- Peter Bell - http://www.pbell.com/
- Ben Garrett

Open Source Code Usage

Many thanks to the following people whom I have used their open source projects. Please visit them and use their software; support open source.

- Blog CFC by Raymond Camden
- Galleon Forums by Raymond Camden
- Zip.cfc by Artur Kordowski
- cfcViewer by Oscar Arevalo
- i18N by Paul Hastings
- Optimization and WS Refresh by Dave Stanten
- Public Issue Tracking & Wiki by Trac
- JavaLoader and Transfer by Mark Mandel
- FileWriter, StringBuffer by Greg Lively
- Illidium PU-36 by Brian Rinaldi
- Brian Kotek's Projects

About the Author

Luis F. Majano

Luis Majano is a Computer Engineer with over 10 years of software development and systems architecture experience. He was born in San Salvador, El Salvador in the late 70's, during a period of economical instability and civil war. He lived in El Salvador until 1995 and then moved to Miami, Florida where he did his Bachelors of Science in Computer Engineering at Florida International University. Luis currently works for ESRI (Environmental System Research Institute) and resides in Rancho Cucamonga, California with his beautiful wife.

He is also the President of Ortus Solutions, a consulting firm specializing in Adobe ColdFusion, Java development and all open source professional services under the ColdBox stack.

He is the creator of ColdBox, Codex Wiki (www.codexwiki.org) an open source enterprise wiki system, and contributes to many open source ColdFusion projects. He is also the Adobe ColdFusion user group manager for the Inland Empire. You can read his blog at www.luismajano.com/blog

Luis has a passion for Jesus, tennis, golf, volleyball and anything electronic.

Random Author Facts:

- I played volleyball in the Salvadorean National Team at the tender age of 17
- The Lord of The Rings is something I read every 5 years. (Geek!)
- My first ever computer was a Texas Instrument TI-86 in 1986. After 1 month, I had written my own tic-tac-toe game in Basic at the age of 9. (Extra Geek!)

Charity Donations

Luis Majano & Ortus Solutions, Corp will donate 20% of the revenues from this book to charity.

Personal Acknowledgments

This book or anything ColdBox would not be possible without God's wisdom and guidance. It is because of His grace that this project exists and the entire honor goes to God alone. If you are offended by these statements or do not like them, then don't read this, it is not for you.

"Therefore being justified by faith, we have peace with God through our Lord Jesus Christ: By whom also we have access by faith into this grace wherein we stand, and rejoice in hope of the glory of God. And not only so, but we glory in tribulations also: knowing that tribulation worketh patience; And patience, experience; and experience, hope: And hope maketh not ashamed; because the love of God is shed abroad in our hearts by the Holy Ghost which is given unto us. ." Romans 5:5

Keep Jesus number one in your life and in your heart. I did and it changed my life from desolotation, defeat and failure to an abundant life full of love, thankfulness, joy and overwhelming peace. As this world breathes failure and fear upon any life, Jesus brings power, love and a sound mind!

Este libro es dedicado a mi esposa Veronica. Gracias por tu paciencia y amor bbita. Es un privilegio ser tu esposo y saber que estamos en esto juntos. Gracias por tantas noches y fines de semana que sacrificamos para poder salir adelante con ColdBox y Ortus. Sos mi inspiración y mi motivación bbita, gracias por entenderme y apoyarme en mi sueño.

About the Technical Reviewer

Kalen Gibbons

Kalen currently develops rich Internet applications for ESRI (Environmental Systems Research Institute) in Redlands, California. He graduated from the California State Polytechnic University of Pomona in 2007 with a degree in Computer Information Systems. He's passionate about programming and enjoys working with many technologies such as ColdFusion, Flex, AIR, and AJAX. He spends his spare time at home with his wonderful wife and two children.

Chapter 1 »
Getting Started With ColdBox

What is ColdBox?

ColdBox is an event-driven, convention, based ColdFusion Development Platform. It provides a set of reusable code and tools that can be used to increase your development productivity, as well as a development standard for working in team environments. ColdBox is comprehensive and modular, which helps address most infrastructure concerns of typical ColdFusion applications. It also goes places that other frameworks do not.

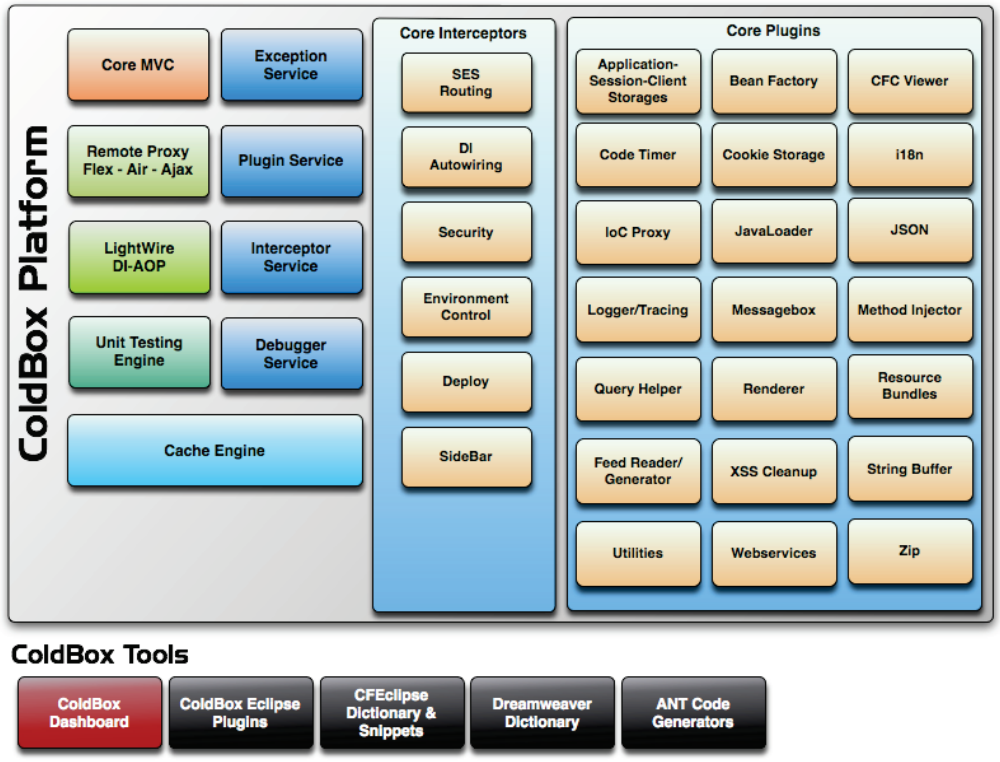


Fig 1.1: ColdBox Platform Diagram

This section will provide an overview of the main components of this object oriented framework. Below are some good resources for you to read about design patterns and other object orientation goodness. Having some basic object oriented knowledge will help you tremendously during your initial stages of ColdBox

development. However, if you are not an OO (object oriented) guru, no worries, the chapters in this book will help you and guide you through several learning paths of object orientation and software development. This is just an introductory section, so you might encounter new terminology or features of the framework that you might have no clue about. However, do not despair, as it will all come clear as you keep reading.

- Sun's Core J2EE Patterns Catalog
    - http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html
- Catalog of Patterns of Enterprise Application Architecture
    - http://martinfowler.com/eaaCatalog/
- What are CFC's by Ben Forta
    - http://www.adobe.com/devnet/ColdFusion/articles/intro_cfcs.html
- ColdFusion CFC Tips
    - http://www.oreillynet.com/pub/a/javascript/2003/09/24/ColdFusion_tips.html

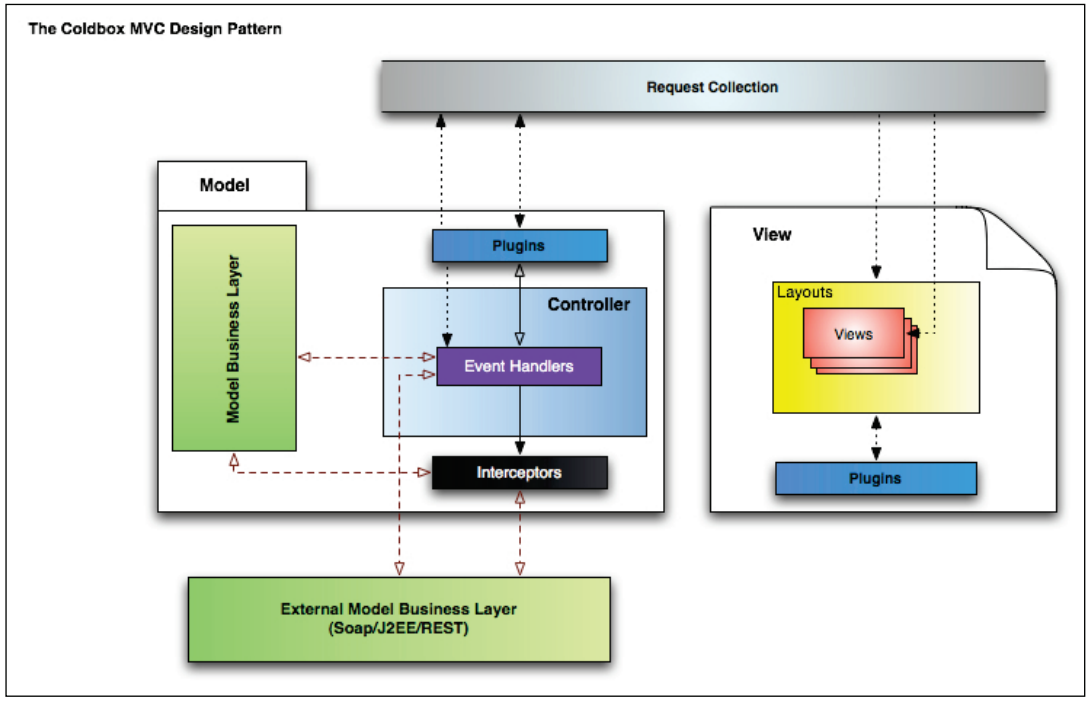


Fig 1.2: ColdBox MVC Design Pattern

What Are Some of ColdBox's Novel Features?

Documentation

As you remember from my introduction, I am a firm believer in developer education. There are over 30 step-by-step online guides, over 550 pages worth of documentation right in the online wiki, 2 professional training courses, and several printed books. It is my belief that by empowering the developer with knowledge, the adaptation rate will increase, and the ability of the developer to find what they need will make their development productivity increase.

Custom Conventions

Conventions over configurations is our motto. We get rid of the verbosity of XML logic and use simple ColdFusion and a set of conventions for our applications. With ColdBox you can even define your own application layouts and conventions a-la-carte. This gives great flexibility to developers or organizations that are used to their own layouts and structures. Conventions are also used for registering events, interceptors, plugins and much more.

The use of conventions over configurations is what makes ColdBox unique!

ColdFusion Controllers instead of XML Controllers

ColdBox doesn't rely on XML declarative logic to define events, what they do and where they go. ColdBox is a conventions based development platform that will let you program in ColdFusion, to get things done faster and easier. You basically expose methods on event handler CFCs (Controllers) by setting their access to public or remote. The framework will auto-register the handler CFCs and you will be able to use the methods as ColdBox Events. So the declarative logic is placed within the methods, where you can place exit points, what model objects to use and call, what view to render, what event to surrender execution, etc; but in ColdFusion and not XML. This is how ColdBox can help you create multi-layered applications with a single skeleton and configuration file. So instead of working with a long and complex configuration file all the time, you will be mostly working with ColdFusion code all the time. You would simply use the configuration file to setup your project or maybe tweak some settings.

Aspect Programming

ColdBox comes bundled with an extensive array of plugins and interceptors that will help you with every day software application tasks like bug reports and notifications, AOP file logging with auto-archiving, per-environment settings, storage facilities for cluster environments, object caching, datasource declarations, web services integrations, internationalization, IoC integrations, application security, SES URL Mappings and so much more. ColdBox is not only an MVC framework but also a development platform.

ColdBox Dashboard

The ColdBox Dashboard is a developer tool that helps you configure your platform installation and has tools for code generation. It is also a self-documenting tool that will help you learn about the framework.

You can modify all of the framework configurations and read documentation. It is tightly integrated to the online documentation so you can search the wiki, svn repository, and ticket reports, and much more.

ColdBox Enterprise Caching

ColdBox has an advanced memory aware and configurable enterprise caching engine. You have several tuning parameters for the cache as well as visual cache reports in the debugging panel. You can actually see how many objects and what types of objects are in your cache, the efficiency of your cache and the tuning parameters. This feature will help developers save time and also provide them with rock solid engine that can manage object persistence. ColdBox also allows for event caching, in which the HTML output events produce will be cached by the framework and presented to users. This will enhance applications and system stability. The best part of it is that you can use metadata in the *cfcomponent* and *cffunction* tags to actually declare caching parameters. ColdBox also allows for extensive view caching and on-demand rendering and caching capabilities. To top it off, the caching engine has an event broadcaster model built-in that can advice you of new objects, object removals, JVM garabage collections and much more.

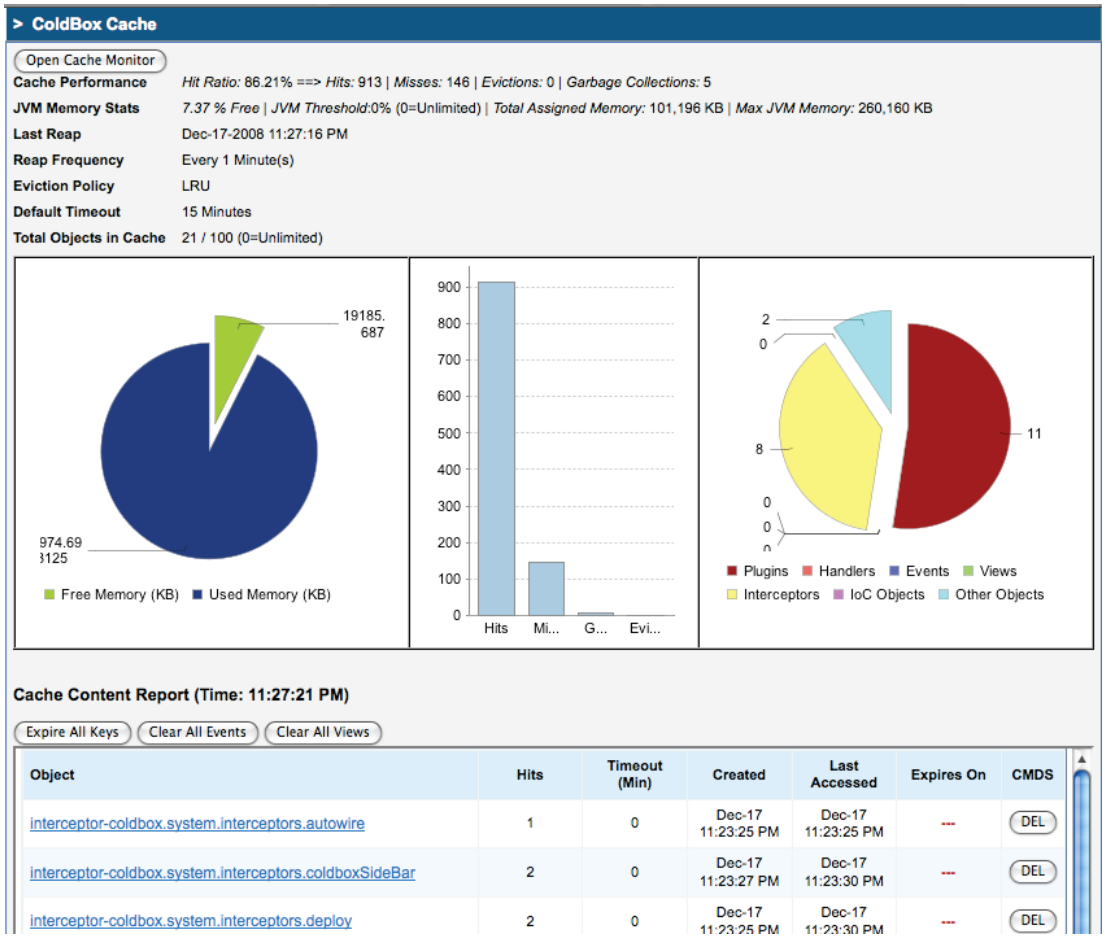


Fig 1.3: Cache Monitor

Unit Testing

ColdBox is a framework based on objects and unit testing is an integral part of development; so why shouldn't you be able to unit test your handlers, plugins and interceptors? Well, Unit Testing is part of ColdBox. ColdBox includes a unit testing feature that allows you to do integration testing, unit testing and even mock objects (3.0 only). It can even provide you with mocking capabilities so you can event test URL relocations and re-routing.

ColdBox Proxy: Flex/Air/Remote Integration

The ColdBox Proxy enables remote applications like Flex and AIR to communicate with ColdBox, providing an event model for those applications. Not only that, but you can reinitialize the entire application, get settings, and yes, announce custom and core interceptions. You can create custom interceptor chains that respond to model calls and they can even be asynchronously. You can create a service layer with built-in environmental settings, logging, error handling, event interception and chaining, you name it, and the possibilities are endless.

Not only that, but also this enables you to create any number of front ends using the same reusable ColdBox and model code. The code is the same, you create event handlers, you interact with a request collection, with core and custom plugins, but you don't set views or layouts because the framework is now a remote framework for your model. So what do you do, well, return data, arrays, XML, and value objects. Anything you want right from within the event handlers, or you can setup a configuration setting that tells the framework to always return the request collection.

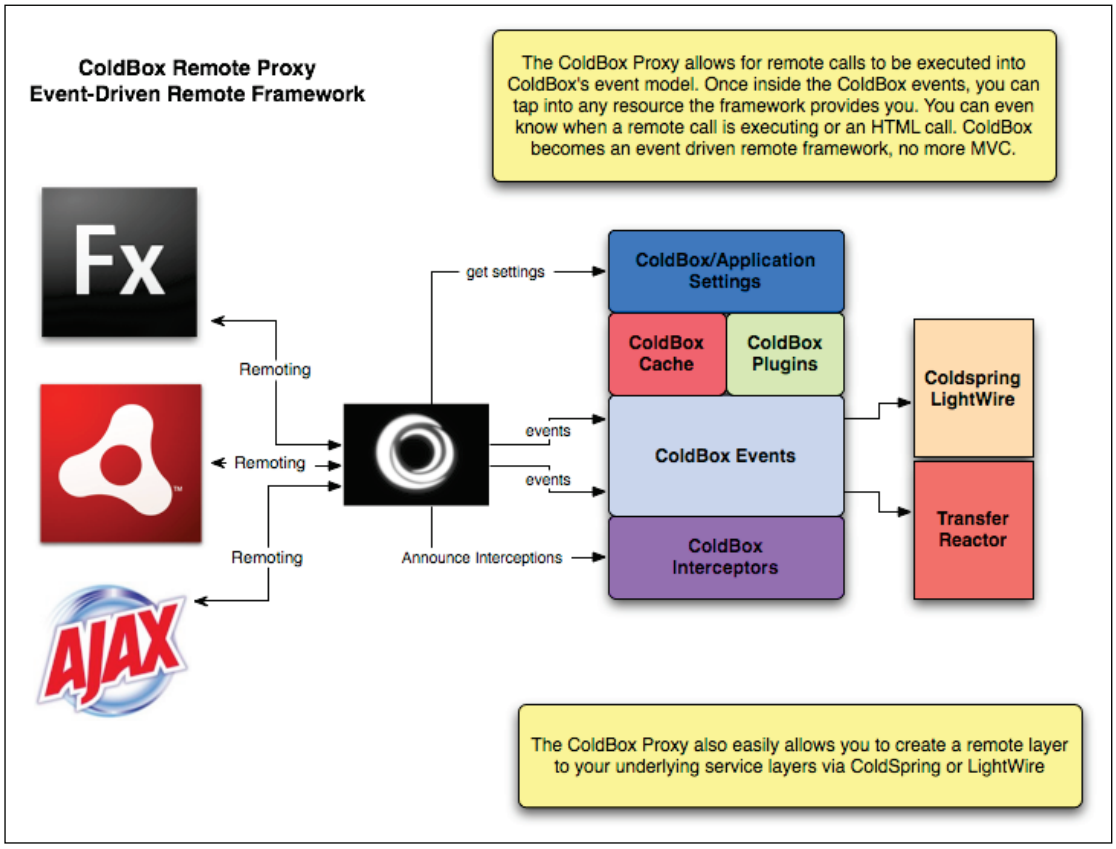


Fig 1.4: ColdBox Proxy Eco System

How ColdBox Works

ColdBox uses both implicit and explicit invocation methods to execute events and render content; ColdBox is an event driven framework. You have a single XML configuration file: *coldbox.xml.cfm*, from which you can configure your entire application (No logic, just configuration data). You can use ColdSpring, Transfer, Remoting, CRUD, Bean/DAO Factories, or any other technology and/or pattern that you can think off with ColdBox. However, ColdBox does make you adhere to an application directory structure based on conventions that are fully customizable. This is done for the purpose of creating a standard for all developers who work in a team and to allow ColdBox to find what it needs. Remember that ColdBox will not solve all your problems. It is a standard, a foundation to develop upon and thanks to the software programming aspects that it provides, ColdBox is also a development platform. However, it is up to you to create GOOD code, this is not a magical framework that will make your code better. It will help you, but at the end of the day, it is your responsibility.

ColdBox makes use of the Front Controller design pattern as its means of operation while in MVC mode. This means that every request comes in through a single template, usually *index.cfm*. Once the framework, through this front controller, receives a request, it will parse the request and re-directit appropriately to the correct execution paths. Events are detected by a URL/FORM/Remote variable named *event* by default.

This event variable holds a specific pattern that lets the framework know what to execute. This is called ColdBox event syntax:

- **NON SES MODE** [handler|package].[action]
- **SES MODE** /index.cfm/{package}/{handler}/{action}

Note: The *index.cfm* file can be removed when using a URL rewrite tool like Apache mod_rewrite.

The handler is the name of the event handler CFC and the action is the name of the public or remote method to execute. You can also prepend package names (directories) of where event handlers can be found. Please note that our recommendation is to use SES routing so you can create meaningful URI's and abstract the real names of the handlers and locations.

"You are only limited by your ingenuity" Luis Majano

Implicit & Explicit Invocations

ColdBox Events can be registered for execution in two different ways. The following events are registered in the configuration file, which are run implicitly (no need for you to call them):

- Request Start Handler (simulates onRequestStart)
- Request End Handler (simulates onRequestEnd)
- Application Start Handler (simulates onApplicationStart)
- Session Start Handler (simulates onSessionStart)
- Session End Handler (simulates onSessionEnd)
- Default Event (The default event to execute)
- onException (The event to execute when an exception occurs)
- onInvalidEvent (The event to execute when an invalid event is detected)

There are also some more methods that will be executed implicitly by writing the following methods inside of a handler CFC:

Method	Description
preHandler	Executes before the requested event (in the same handler CFC)

postHandler	Executes after the requested event (in the same handler CFC)
onMissingAction	Executes if an action was requested from this handler but the method does not exist (In the same handler CFC)

The other approach to executing events is via explicit declarations from within your code using the *runEvent()* method. From these events, you declare what business actions to invoke, what view to render, call/use plugins, and if more events need to be executed (chaining). All these actions are done explicitly; you define them in CF code and **not in XML dialect**. The ColdBox controller then implicitly renders layouts/views that were set by the event handlers and finalizes execution.

One thing to note is that the event handler's events (*methods*) are very loosely coupled to each other. They interact on their own, do what they need to do and surrender execution to the framework. As you can see, due to the nature of event handlers written in ColdFusion, you have explicit declarations that would otherwise be implicit if done in an XML based dialect. Thus, the cohesion between implicit and explicit event executions can exist in a ColdBox application. At the end of the day, ColdBox is based on events and cannot function without them.

Configuration File (coldbox.xml)

ColdBox is configured for operation via a single XML file. You can define the major settings for your application, what features to use, etc.

Event Handlers (Controllers)

ColdBox event handlers are CFC's that act as your application controllers. Most of this topic is covered in the Chapter 9, but here is a brief introduction:

- First of all, these handler CFC's must extend the ColdBox base event handler CFC: *coldbox.system.eventHandler*
- If the CFC implements an *init()* method, then it must call the base class constructor using the *super* method. Remember that if you implement an init method in an event handler, then all the methods executed from this handler will run the constructor code.
- The CFC's must be placed in the *handlers* directory under your application, so they can be registered by conventions.
- You need to create public/remote methods that will respond to events.
- The framework will cache event handlers, and cache metadata can be attached to their *cfcomponent* declaration.

On initialization the framework will read your *handler's* directory and register the available handlers. Once an event is detected, the framework will validate both the handler and the method. Therefore, in order to expose an event to the framework, just create a method in your CFC with public or remote access.

Event handlers can also be called from remote applications such as Ajax and Flex via the ColdBox Proxy. You can even determine if an incoming request is an MVC or remote request and react accordingly. In remote mode, your event handlers can return data instead of rendering views.

Request Collection

ColdBox also uses a request collection data structure where all variables can be stored and shared among an execution request. The request collection is a central repository of information that is refreshed on every user request with the request's information. This is how data gets moved around from event handlers to views and layouts to plugins and anything running inside of the framework and in the MVC layers. The object containing the request collection is the *request context objec*t found at *coldbox.system.beans.requestContext*. Not only can you use the request context object but also you can decorate it!

You can expand its functionality according to your needs; refer to Chapter 8 to help you learn more about how to extend the core framework's functionality. The request context is not a mere data structure where you get and set values, but an object that is used for setting views or layouts for rendering, caching views, getting event metadata, determining what is being executed and so much more.

Plugins

Another important ColdBox feature is the use of a plugin library of CFCs that extend the normal usage of ColdBox to application specific tasks, without hindering system performance. These plugins are reusable components that your application can use and can be loaded on demand via the ColdBox Plugin Factory. Some samples are: i18n, resource bundles, refresh a webservice stub, Bug Reports, Java file utilities, etc. This is a major difference between ColdBox and any other framework, in that it gives you a set of reusable on-demand components for tedious or repeatable application specific tasks.

Also, not only can you use and modify the plugins that come with every ColdBox installation, but also you can create your own. You are not limited to what we provide, you can extend the framework to meet your needs. You can create as many plugins as you want and build a plugins library that can be registered by just specifying it on the configuration file or can be loaded purely by conventions . You will then be able to get the plugin and use it on any of your ColdBox applications. The best part of it all is that the plugin will inherit all of the framework's functionality, so you have everything that you need to be able to code.

Interceptors

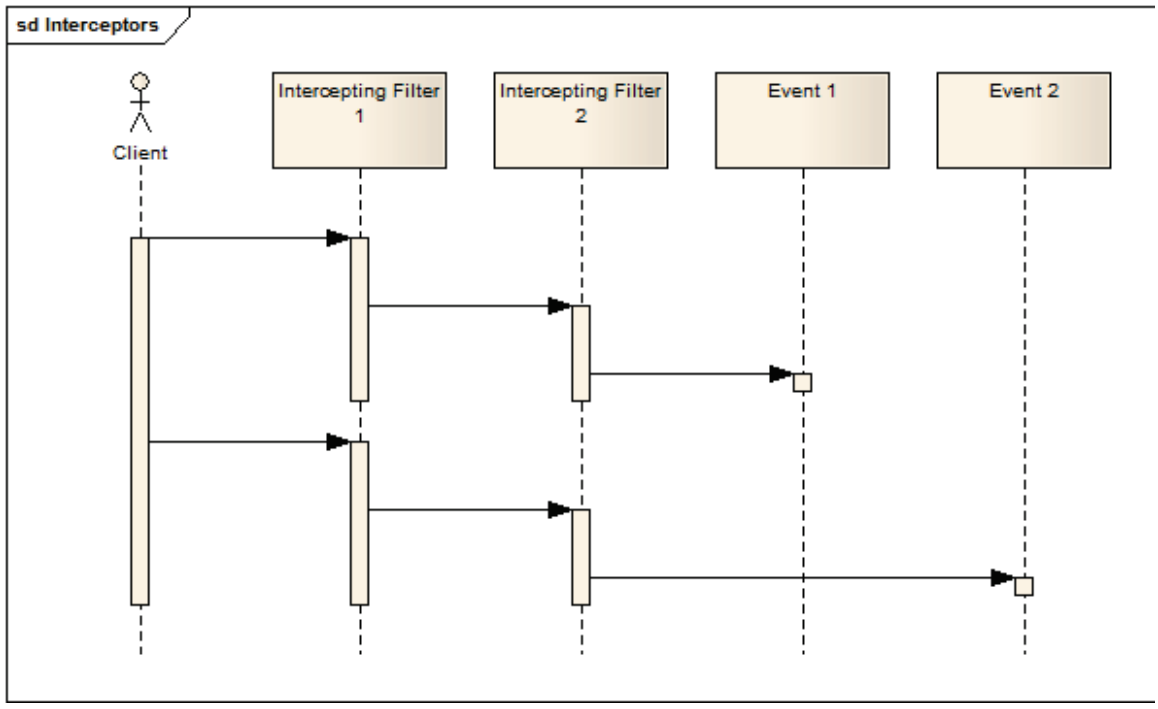


Fig 1.5: Interceptor Sequence Diagram

ColdBox interceptors increase functionality for applications and framework alike, without touching the core functionality, and thus encapsulating logic into separate objects. This pattern wraps itself around a request in specific execution points in which it can process, pre-process, post-process and redirect requests. These interceptors can also be stacked to form interceptor chains that can be executed implicitly. The chaining is all about positioning in the configuration file. The order of declaration is very important. These stacked interceptor chains form a sequence of separate, declaratively deployable services to an existing web application or framework without incurring any changes to the main application or framework source code. This is a powerful feature that can help developers and framework contributors share and interact with their work.

Interceptors are a great compliment to ColdBox plugins, they can be used alongside them and implicitly add functionality to a ColdBox application. Another important aspect to note is that interceptors have full access to a request's lifecycle and the framework. Thus, they can get application settings, redirect control, execute events, use the cache manager, get plugins, transform views, adapt views for certain protocols and much more.

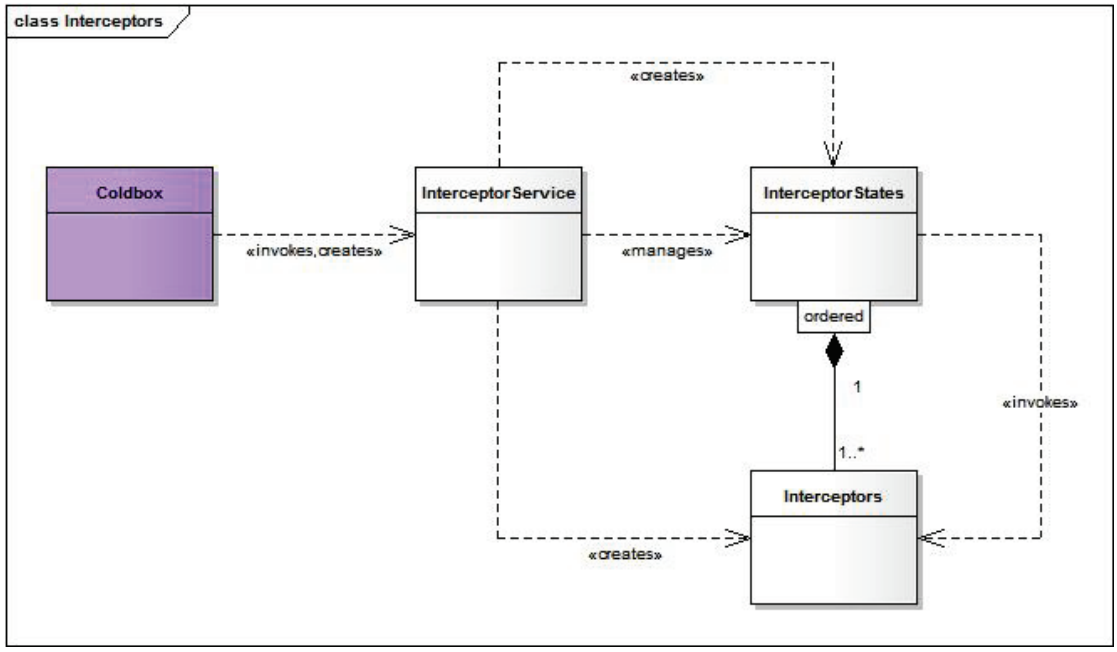


Fig 1.6: Simple Interceptor Overview

Interceptor Applications

Below are just a few applications of ColdBox Interceptors:

- Event based security
- Method tracing
- AOP interceptions
- Publisher/Consumer operations
- Implicit chain of events
- Content appending or pre-pending
- View manipulations
- Custom SES support
- Cache advices on insert and remove
- Much more...

Much More Than Interceptors (Observers)

We went a step further with interception points and created the hooks necessary in order to implement an observer/observable pattern into the entire interceptor service. What does this mean? It means that you are not restricted to the predefined interception points that ColdBox provides; you can create your own WOW!

Really? Yes, you can very easily declare execution points via the configuration file or register them at runtime; create your interceptors with the execution point you declared (conventions baby!) and then just announce interceptions in your code via the interception API.

"The power of conventions in its full strength." Luis Majano

However, not only can you intercept at an execution point, but you can actually send a structure of intercepted data right into the interceptor. You can use these custom interceptors as a set of listeners waiting for broadcast information. You can even create chains of listeners for the same message.

Summary

I hope that this overview has given you an insight into how powerful ColdBox is for building your web applications whether they are small or enterprise. It is a new generation framework based on conventions that will increase your productivity and adaptability in a team environment.

Welcome to the ColdBox Platform!

Chapter 3 »
ColdFusion Components & OO Terms

Before we start devling into the nitty gritty of ColdBox, in this chapter I would like to focus on the basics of ColdFusion Components and some object oriented terminology. This section will cover what we (Team ColdBox) believe to be best practices for developing components (objects) in ColdFusion. It will also introduce some great object oriented terms so we can all be on the same level once we start digging further into ColdBox. So let's begin.

What is an Object?

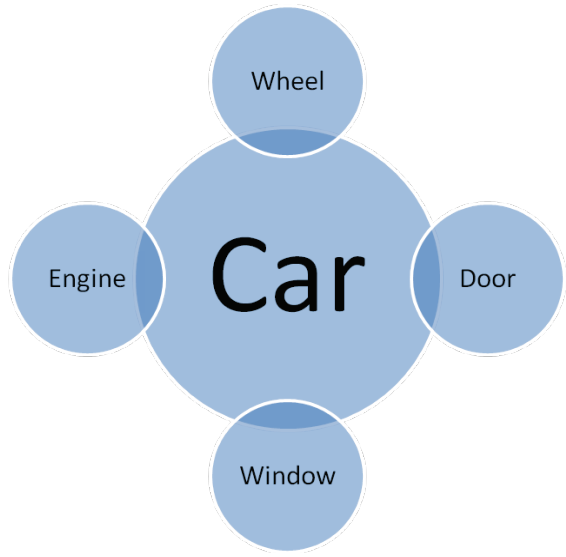


Fig 3.1: Simple Object Representation

The object oriented paradigm centers on the concept of an object and not functions. Functions are what an object provides in order to interact with it. In its core, an object is a state of being that has an identity, a purpose and a set of responsibilities. Think of objects as nouns in the bare minimum, such as students, courses, instructors, etc. The key to good object orientation is to know an object's identity or state of being. This is commonly called as *Ontology* or the study of *Ontology*.

Ontology describes or asks, what constitutes the identity of an object?

So as you start analyzing and creating object oriented applications, you always have to keep in mind what are the object's responsibilities and most importantly, the object's identity. A good design rule is that an object should be responsible for itself and should have its responsibilities very clearly defined. At the implementation level, an object is purely code and data.

What is a CFC?

A CFC is short for ColdFusion component. A CFC is the way to represent objects programmatically in ColdFusion. It has a set of conventions, tags and syntax that will enable you to program objects. A CFC is declared by the *cfcomponent* tag, and methods are defined with the *cffuntion tag.* We will not cover all the details about these tags here; we are just covering the best practices for programming ColdFusion components as we do it in the ColdBox team.

```
<cfcomponent name="Student" output=false hint="My student object">

<cfscript>
  instance = structnew();
  instance.name = "";
</cfscript>

<cffunction name="getName" returntype="string" access="public">
  <cfreturn instance.name>
</cffunction>

<cffunction name="setName" returntype="void" access="public">
  <cfargument name="name" type="string">
  <cfset instance.name = arguments.name>
</cffunction>

</cfcomponent>
```

Object Oriented Terms

Let's review some object oriented terminology so that we can get into that learning mood.

- **Class**: ColdFusion CFCs : A blueprint of an object. Typically they have methods/functions and instance data.
- **Constructor**: A method responsible for object creation and for preparing an object for usage.
- **Attributes**: Instace data associated with an object (data members). They can also have an access type or visibility.
- **Access Type**: A visibility property for attributes and functions, if exposed as public then it becomes part of the object's public API. Usual visibility types in ColdFusion are *public, private, package, remote.*

- **API**: *Application Programming Interface*. An API implements information hiding by its exposed (public) members (functions) and attributes (properties).
- **Methods**: Functions associated with an object. If they are exposed as public they become part of the object's API.
- **Accessors/Mutators**: Accessors are usually referred as "getters" or "get methods" that retrieve instance data. Mutators, as the word delineates, changes the instance data via a "set method" or "setter".
- **Encapsulation**: Information hiding by usage of methods and their visibility types. It is also the act of creating methods for anything that varies within your objects. *DRY* (Do not repeat yourself). Makes writing code easier as external forces only interact with the object's public API and do not need to know how the object is written or its internal properties and methods.
- **Instance**: A particular object of a class (component); creating a class or instantiating a class object.
- **Derived Class (Inheritance):** A class that is specialized or derived from a parent class representing an "is-a" relationship or taxonomy. Ex: Cat "is an" Animal. The derived class contains all the properties and methods of the parent class but can also contain more properties and methods.
- **Composition**: Object composition exists when a class has another class for a property or part of its instance data. The composite class often handles the creation of the composite parts. A car class can have a composite of engine or multiple tires, for example. Composition denotes a tight coupling between the classes, if the class composed of other classes is destroyed, all the composite classes are destroyed also.
- **Aggregation**: Follows the same principle as object composition with the distinct difference that the composite part's lifespan is NOT controlled by the agregee. For example, a parking lot object is composed of 0 or more car objects. However, if we destroy the parking lot, the cars do not get destroyed.
- **Design Patterns**: A time-tested architectural solution to a recurring problem. They do not solve all problems but are great references for common solutions.
- **Coupling**: The degree in which each program module relies on each one of the other modules in a system. We want low coupling in order for modules to not have tight dependencies to other modules. Why? Well, one small change in a tighly coupled class could wreck havoc in an entire set of other classes. The more we decouple our code, the more sustainable our applications will become.
- **Cohesion**: Measures of how strongly related or focused are the responsibilities of an object. An object must do one thing and do it right. We want high cohesion. Why? If not, our objects will start becoming GOD objects, where they do everything or more than they should. This introduces complexities and makes maintenance more difficult.
- **DSL**: Domain Specific Language is a programming language or specification dedicated to a particual problema domain, representation or technique.

ColdFusion Best Practices

We will start covering all the best practices for ColdFusion components in this section in no particular order. Please note that almost all of ColdBox's interactions are via CFC. Therefore, it is essential and necessary to present several best practices for CFC's that will apply to almost all of ColdBox application development. You can find a more in depth article about best practices online in our wiki. Our best practices shown here are a composition of best practices from the ColdBox Team and many online resources:

- http://ortus.svnrepository.com/coldbox/trac.cgi/wiki
- Sean Corfield's CFC Best Practices Document
    - http://www.corfield.org/
- http://www.adobe.com
- http://cfdj.sys-con.com/read/41660.htm

Note: At the time of writing our wiki was in transition to http://wiki.coldbox.org where you can find our latest guides and information.

Constructor

ColdFusion components should always have a method called *init()* that will act as your constructor. This is how your CFC is initialized and prepares your instance for usage; it should always returns the *this* scope. By returning the *this* scope, you are returning the instance of the CFC as a reference object to the caller object or page.

```
<cffunction name="init" output="False" returnType="{filename}" hint="">
  <cfreturn this>
</cffunction>
```

Single Responsibility

Single responsibility means that your CFC must adhere to its responsibilities ONLY. If you find yourself with a CFC with over 500 lines of code, then it might be of good wisdom to determine if it is doing more than it should. If it is, then you need to separate or refactor your code into other objects. There are very rare cases where an object can get this long and just do one set of responsibilities, but it is possible. Good judgment should be used.

If you find yourself writing methods that are over 30-50 lines of code, you might also want to revise your work. You might find that a method is doing too much and it needs other methods to help out.

Attribute Scopes

The two major visibility scopes that ColdFusion gives us inside of components are:

- **variables** : private scope
- **this** : public scope

```
<cfset address = CreateObject("component","address").init()>
<cfset variables.address = CreateObject("component","address").init()
```

Both lines of code above are the same. As the *variables* scope is the default scope in a page and CFC, so it is not necessary to scope it if you so desire.

It is also very important to realize that these scopes are not only used for attributes but also for methods. The ways that ColdFusion stores public methods are by placing them in the *this* and *variables* scopes. If you write a private method, then ColdFusion places it in the *variables* scope only. If you write a public method, then ColdFusion places it in both the *variables* and the *this* scopes. Therefore, it is very important to realize what these scopes do and what they represent.

```
<cfset this.OPTIONS = "add,remove">
<cfset this.NOT_FOUND = '_NOTFOUND_'>
<cfset this.EVENT_CACHEKEY_PREFIX = "cboxevent_event-">
```

Public variables are declared in the *this* scope. Be very careful of when to make internal properties public, as you will be violating encapsulation (look at next sections). One of the best reasons for making variables public is if they do not change or can act like static constants. If your variable does not meet this criteria, then DO NOT expose it as public. However, please remember that this is a guideline and not a rule. Also note that if you declare an attribute as public, it can be easily changed by an outside force. You have no control of the data of this variable or how it will be used. Therefore, if you do not need to expose it, then hide it. This is also referred to as information hiding. Outside forces DO NOT need to know how you store your data, they are concerned with what you offer via your public methods (API)

Virtual Scope: Instance

A good best practice is to create a virtual scope as a private attribute of a CFC. You can do this in the CFC's pseudo constructor (space between the *cfcomponent* tag and the first method) or constructor (*init*() method). A sample is found below:

```
<cfset instance = structnew()>
<cfset instance.firstname = "Luis">
<cfset instance.lastname = "Majano>
```

The purpose behind this approach is to create a structure holder for all the instance data that is separate from the CFC's methods. This is also a real benefit when implementing a Memento pattern or State pattern where all instance data of an object can be serialized or deserialized into an object. You can very easily move the state of this object to another object by implementing some memento methods:

```
<!--- Getter/Setter memento --->
<cffunction  name="getmemento" access="public" returntype="struct"
             output="false" hint="Get the memento">
  <cfreturn variables.instance>
</cffunction>

<cffunction  name="setmemento" access="public" returntype="void"
             output="false" hint="Set the memento">
  <cfargument name="memento" type="struct" required="true">
  <cfset variables.instance = arguments.memento>
</cffunction>
```

Encapsulation

Encapsulation provides the basis for object orientation by providing information hiding from the outside world for an object. Encapsulation is achieved by designing objects with public methods that expose functionality rather than direct manipulation of the object's internals via properties or attributes. Encapsulation is achieved by declaring access types of variables as *private*. This gives access to data to only public/package member functions of the CFC. You can then create their mutators and accessors via "*public*" methods. Some benefits of encapsulation are:

- You can keep the exposed API the same while changing how your CFC works internally without breaking any code that uses your CFC. You can refactor your code with ease.
- Prevents the consuming developer from getting in trouble by violating your internal assumptions about how the instance data works, since all functionality is exposed as methods.
- It hides all of your implementation and presents a good public API.
- Writing code is easier as team members only have to interact with this API instead of how the object works under the hood.
- As a rule of thumb, encapsulate anything that varies.

DRY: Do not repeat yourself!

Referencing External Scopes

Do not directly refer to external scope variables *(i.e.: session/application/client/server/form/url/request variables)* inside a CFC, especially in ColdBox. When in doubt, preserve encapsulation. The one exception is when building facades or proxy objects for web services/flash remoting or special encapsulations. If you do not know what a façade or proxy is, then please do a web search for "façade or proxy pattern". It basically encapsulates a shared scope such as application, session, etc in an object. Why? If you reference variables in a CFC you are binding or coupling yourself to that scope and variable existence from within your object, when your object might just need that data passed throught to it. Maintenance will increase exponentially and unit testing these objects become harder to do.

Always Use OUTPUT=false

Use the *output* attribute in your *cffuntion* and *cfcomponent* tags. Do not output directly to the output buffer inside a CFC method - instead return a string. The main reason is that you don't want to break encapsulation and be outputting content from all kinds of places. By outputting directly to the output stream you assume knowledge of the external environment of the CFC, but if you return a string you get the exact same behavior without breaking encapsulation. Never assume that the CFC will run when you want it to run, especially when dealing with threaded requests.

```
<cfoutput>#myCFC.getSomeHTML()#</cfoutput>
```

On very rare cases you would want to output to the buffer, but avoid it at all costs. As a last note, if you do not use *output=false* on *cfcomponent* and *cffunction* tags, you most likely will be producing the dreaded ColdFusion whitespace in your rendered HTML. So make sure you use this attribute to prevent unnecessary whitespace in your rendered HTML.

Naming Conventions

Naming conventions are not strict but sure make life easier when everything that you read makes sense. Use good names for components, methods, arguments and local variables. This can sometimes be a disaster if developers choose random names or non qualified names for methods, arguments and local variables. Most of the naming conventions that we follow in Team ColdBox are those designed by the Java Community.

- Camel case your component names and capitalizing the first letter: *UserService.cfc, HTTPFacade.cfc, SessionService.cfc*. If your component name uses acronyms, make sure that you capitalize all of them, e.g. HTTP, URL, etc. However, try to avoid them if necessary.
- Try to keep your component names simple and descriptive.
- Interfaces should be capitalized and be used just like component names.
- Camel case your methods and arguments without capitalizing the first letter: *isOpen(), getString(), authenticateUser(), tokenStream()*. Try to use verbs when possible.
- Camel case your instance data members without capitalizing the first letter just like methods and arguments.
- Try to ALWAYS name your variables in a descriptive and non-cryptic manner.
- Although constants do not exist in ColdFusion, you can distinguish them by UPPER casing them.
- Lower case your package (directory) names if at all possible.

Use Return Types

Use the *returntype* attribute of *cffunction* and the *type* attribute of *cfargument* to create documentation and for runtime type checks. Don't forget that "*void*" is the *returntype* when not returning anything from a method.

Duck Typing

Duck Typing is used by setting the *returntype* or *type* to *any*. This is a useful technique when dealing with such a dynamic language as ColdFusion. This means that the argument or returntype returned can be *ANYTHING*, so your object needs to determine what to do with it and what the object's identity is based on preset conventions. This is what makes a dynamic language like ColdFusion so powerful; we do not have to rely on type checking in order to program. However, with so much power, we must have responsibility because now we do not get the benefit of compile time checks, so we need to do runtime checkings. This is why unit testing is SO important when building dynamic objects in ColdFusion.

A side effect of not using a strong type is a speed enhancement, since ColdFusion does not check the validity of the types. This side effect should not be used to get more performance, unless you really are desperate for it. Also note that by providing a *returntype* or *type* of *Any*, you will loose all documentation for it; so make sure to include hint attributes. So if you are building publicly consumed APIs or libraries, try to avoid Duck Typing unless you want objects or arguments to be dynamically determined at runtime.

Use Hints

Document your component, methods and arguments by using the *hint* attribute in those tags. This will help fellow developers and even you, when determining what a method, argument or component can do, etc. You can also use several tools to create CFC documentation according to your component's metadata. The ColdBox team tends to always use ***ColdDoc*** by Mark Mandel (colddoc.riaforge.org).

Use Default for Non-Required Arguments.

In general, non-required arguments of a CFC method should have a *DEFAULT* specified unless you will be determinining them via existence, which can also be a good strategy. However, we always try to have defaults if possible.

Var Scoping

Always, always, always use "*var*" for local variables inside your methods, including ALL loop counters, temporary variables, queries, cfhttp variables, etc. This is called *"var scoping"*. If you do not do this, your component will not be thread-safe. This means that if somebody persists (stores) this component in memory, succinct calls can and will override variables and create all sorts of memory problems.

There is an open source project called ***varscoper*** that can check all of your components for var scoping issues, even if they are using cfscript. (varscoper.riaforge.org)

```
<cffunction name="myFunction" access="public" returntype="void" output="false"
hint="This methods does nothing">
  <cfset var i = 0>
  <cfset var qGet = 0>
  <cfquery name="qGet">
  Select * from test
  </cfquery>
  <cfloop from="1" to ="20" index="i">
```

```
  <!--- Do Something --->
  </cfloop>
</cffunction>
```

I cannot stress the importance of var scoping your variables. Especially when creating variables in your ColdBox handler CFCs, you will go crazy trying to find how variables change by themselves.

VAR SCOPE EVERY VARIABLE!

Use Inheritance Sparingly

Use inheritance only when describing an *"is-a"* relationship, not for a *"has-a"* relationship or for code reuse. For a nice summary, visit http://cnx.rice.edu/content/m11709/latest/. If you need an object to reuse tons of methods **PLEASE** do not look to inheritance immediately. Take your time and evaluate the situation. Inheritance is fragile and extremely brittle as changes affect an entire tree of classes at compilation and not at runtime. If you want to bring in behavior and functions into an object at runtime, then look to object composition.

Use Inheritance when you can justify taxonomy of components.

Composition Over Inheritance

Always prefer object composition over inheritance. This is where another component is created or injected as a property of the object at hand. There are several reasons why you should choose composition over inheritance. Some resources are:

- http://brighton.ncsa.uiuc.edu/prajlich/T/node14.html
- http://www.artima.com/lejava/articles/designprinciples4.html
- http://guidewiredevelopment.wordpress.com/2009/02/05/favoring-composition-over-inheritance/

The underlying argument is that composition brings in functionality or behavior at runtime instead of at compilation time. This makes your objects much more sustainable and flexible as they don't have to rely on compilation time restrictions.

Arguments By Reference and By Value

Variables pass in and out of components by reference or by value. "By reference" means that if the variable is changed inside of the component call then the changes are also reflected outside of the component call. Basically, the variable is a reference or pointer to the value in memory. "By value" means that the variable's value is passed and not its reference. Therefeore, the value can change without affecting the original variable value. For instance, strings, arrays, numbers, and dates all pass by value, but structures, queries, and all other "complex" objects (including CFC instances) pass by reference. Important caveats to

notice about the types that are passed by value are that arrays are also passed by value. To us, passing arrays by value is completely unnecessary as in Java they are by reference. However, this is how Adobe ColdFusion has implemented them and you need to understand that this might not be the case in other CFML engines or languages.

Important: Please also note that CFML engines like Railo pass arrays by reference instead of by value.

Unit Testing

Unit testing is essential when building object oriented applications because it allows you to test individual components and make sure that they work without constructing entire object graphs. There are great tools out there to help you test CFCs like:

- **MockBox** – The ColdBox mocking/stubbing framework (Included with the ColdBox Platform 3.0.0)
- **MXUnit** – http://www.mxunit.org
- **cfcUnit** - http://www.cfcunit.org/cfcunit/
- **cfUnit** - http://cfunit.sourceforge.net/

Do not be afraid of testing or creating tests, they are rather easy. Get out of your comfort zone and just do it. What will I get in return, you might ask? Well, I can't promise that your code will be better, but it will sure be more robust. You can also create tons of suites and automate your testing before deploys, this assures you (almost 90%) that your code will work as it should once deployed.

Unit testing will also introduce you to more tools for your arsenal and even encourage you to use other technologies like ANT, Maven or other deployment mechanisms that may make your life much easier.

Summary

I hope that you now have a better understanding of components, objects and their best practices. ColdBox extensively uses ColdFusion components and object oriented terminology and patterns. Therefore, we encourage you to take time and learn new technologies and methodologies. You will see the power of object orientation and how your applications will become more extensible and maintainable.

Welcome to OO & let the journey begin…

Chapter 14 »
Model Integration Guide

What is model integration? Well, model integration is an easy and maintainable way of creating, retrieving and using domain model objects within a ColdBox application. The ColdBox team really saw that developers needed an easy and powerful way of accessing domain model objects that would not sacrifice performance, ease of use and adaptability. We were hesitant about doing model integration for quite a while, but we saw all the benefits that it could bring. Not only that, as this book is written, the original model integration code is being abstracted into its own standalone service called *BlenderBox*. This service will provide all of the model integration features, inversion of control, dependency injection, annotations, AOP, etc to any ColdFusion based application; it won't even require ColdBox MVC if it so desired. However, by using ColdBox MVC you get many more benefits. So you can consider model integration to be ColdBox's own internal inversion of control framework but with what we know, love and cherish: conventions.

Again, model integration helps you create, manage, and use model (business logic) objects very easily within any ColdBox application. You will find that the model integration can also be used alongside object caching or even object factories like ColdSpring and LightWire. However, the main purpose for model integration is to make developer's development workflow easier and faster! And we all like that Easy button!

This integration will give you a good kick start on dependency injection, caching, persistence, etc without you learning an XML declarative language. Some very simple conventions are all you need to get you started. Now, what does model integration do for you?

- Easily create and retrieve model objects by using one method: *getModel()* from handlers, plugins and interceptors
- Easily handle model dependencies by using *cfproperty*, constructor argument conventions or old-fashioned setters. In other words, we have our own dependency injection framework based on conventions.
- A conventions *DSL* (Domain Specific Language) has been created in order to facilitate what objects/data needs to be injected in the models (don't shiver with fear yet, please keep reading)
- *Persistence*: use the same rock solid ColdBox cache to persist model objects by using CFC cache metadata. You can now have services that can adjust according to available memory
- Easily create model mappings or aliases for any model class
- Easily populate model objects with data from a request: *populateModel()*

Model Layer Overview

The model layer represents your data structures and business logic. A good definition of a domain model is

> *The domain-specific representation of the information that the application operates on.*

Many applications also use a persistent storage mechanism (such as a database) to store and retrieve data. MVC does not specifically mention the data access layer because it is understood to be underneath or encapsulated by the model layer. This is the most important part of your application and it is usually modeled by ColdFusion components. You can even create the entire model layer in another language or physical location (web services). All you need to understand is that this layer is the layer that runs the logic show! For the following example, I highly encourage you to also do UML modeling, so you can visualize class relationships and design.

A simple example can be described like so. Let's say that you want to build a simple book catalog and you want to be able to do the following:

- List how many books you have
- Search for a book by name
- Add Books
- Remove Books
- Update Books

Very straight forward, right? Anyways, you want to apply best practices and use a service layer approach for your application and model design. You will then use these service objects in your handlers in order to do the business logic. Repeat after me: I WILL NOT PUT BUSINESS LOGIC IN EVENT HANDLERS!

The whole point of the model layer is that it is separate from the other 2 layers (controller and views). Remember, the model is supposed to live on its own and not be dependent on external layers (*Decoupled*). From these simple requirements I will create the following classes:

- *BookService.cfc*
- *Book.cfc*

Service Layer

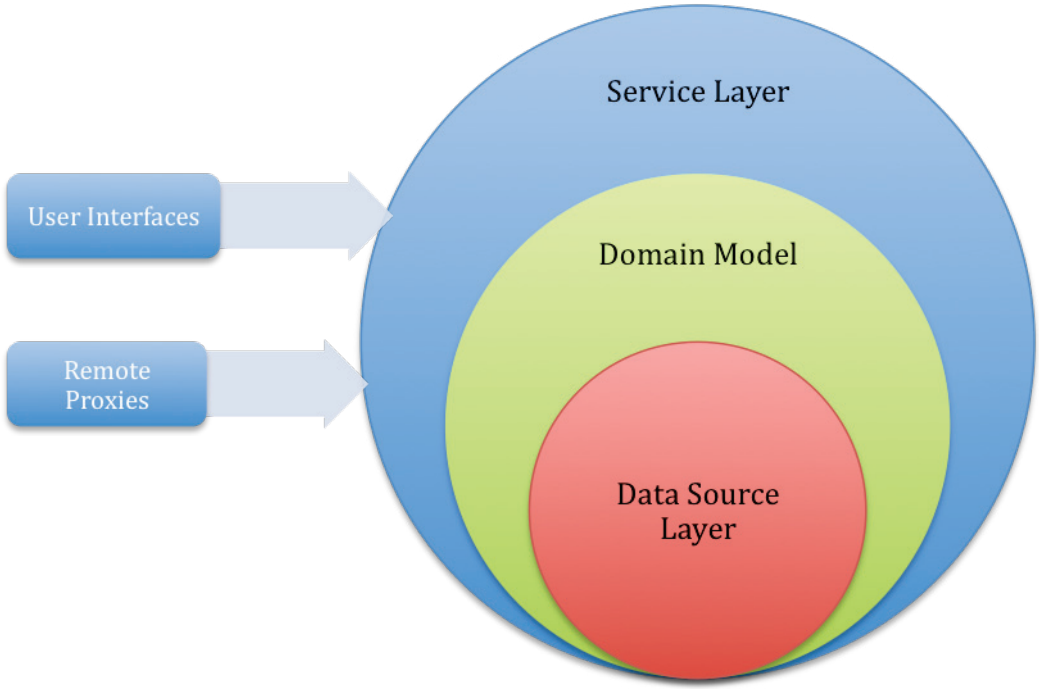


Fig 14.1 : Service Layer Diagram

> *"A Service Layer defines an application's boundary [Cockburn PloP] and its set of available operations from the perspective of interfacing client layers. It encapsulates the application's business logic, controlling transactions and coordinating responses in the implementation of its operations." by* ***Martin Fowler***

A *service layer* approach is a way to architect enterprise applications in which there is a layer that acts as a service or mediator to your domain models, data layers and so forth. This layer is the one that event handlers or remote ColdBox proxies can talk to in order to interact with the domain model. I won't go deep into service layer design or approaches as there are various considerations and opinions on what exactly to put on them or how to layer them. I want to concentrate and challenge you to try these approaches out and learn from your experiences. I believe there is NO SILVER BULLET on OO design, just stick to best practices and practice code smell. Code smell is a term to describe an instinct that you may develop once you start building your applications. This instinct usually tells you when code is written in a bad manner or might be problematic in the future. Consider it as your coding instincts.

The BookService object will be my API to complete the operations mentioned in my requirements and this is the object that will be used by my handlers. My *Book* object will model a book's data and behavior. It will be produced, saved and updated by a *BookService* object and will be used by event handlers in order to populate them with data from the user. The view layer will also use the *Book* object in order to present the data. The event handlers are in charge of talking to the domain model for operations/business logic, controlling the user's input requests, populating the correct data into the *Book* model object and making sure that it is sent to the *BookService* for persistence.

Now, if I know that my database operations will get very complex or I want added separation of concerns, I could add a third class to the mix: *BookGateway.cfc* that could act as my table gateway object or data access object, take your pick. Now, there are so many design considerations, architectural requirements and even types of service layer approaches that I cannot go over them and present them. My preference is to create service layers according to my application's functionality (encompassing one or more persistence tables) and create gateways-DAO when needed. Please also note that I consider a gateway or DAO to be interchangeable. It just means that it is a data layer, don't go creating a gateway and DAO object for a table, this will create an anemic model. The idea is that you have a data layer, that you call it is your business. I usually call them Data Access Objects (DAO).

The important aspect here is that I am thinking about my project's OO design and not how I will persist the data in a database. This, to me, is key! Understanding that I am more concerned with my object's behavior than how will I persist their data will allow you to concentrate your efforts on the business rules, model design and less about how the data will persist. Don't get me wrong, persistence takes part in the design, but it should not drive it.

So what can *Book.cfc* do? It can have the following private properties:

- name
- id
- createdate
- ISBN
- author
- publishDate

It can then have getters/setters for each property that I want to expose to the outside world, remember that objects should be shy and only expose what needs to be exposed. Then I can add extra functionality or behavior as needed. You can do things like:

- Have a method that checks if the publish date is within a certain amount of years
- Have a method that can output the ISBN number in certain formats
- Have a method that can output the publish date in different formats and locales
- Make the object save itself or persist itself (active record)
- And so much more

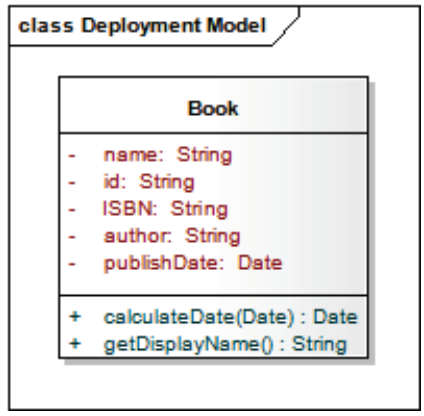


Fig 14.2 : Book Model Object

Now, all you OO gurus might be saying, why did he leave the author as a string and not represented by another object. Well, because of simplicity. The best practice, or that code smell you just had, is correct. The author should be encapsulated by its own model object *Author* that can be aggregated or used by the *Book* object. I will not get into details about object aggregation and composition, but just understand that if you thought about it, then you are correct. Moving along... Your objects are not always supposed to be dumb, or just have getters and setters (Anemic Model). Enrich them please!

Let's go back to the *BookService* object. This service will need a datasource name (which could be encapsulated in a datasource object) in order to connect to the database and persist data. It might also need a table prefix to use (because I want to), which comes from a setting in my application's configuration file. Okay, so now we know the following dependencies or external forces:

- A datasource (as an object or string)
- A setting (as a string)

I can also think of a few more methods that I can have on my *BookService* object:

- **getBook**([id:string]):*Book*
  This method will create or retrieve a book by id
- **searchBook**(criteria:string):*query*
  This method can return a query or array of books if needed
- **saveBook**(book:Book):*void*
  Save or Update a book
- **deleteBook**(book:Book):*void*
  Delete a book

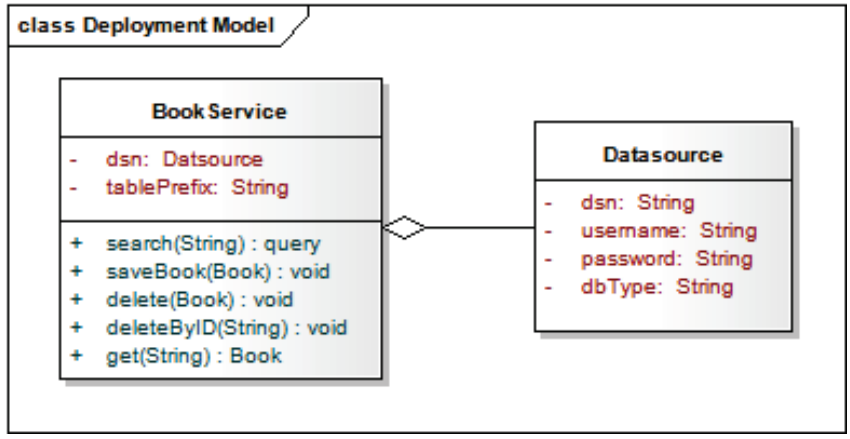


Fig 14.3 : BookService Model

I recommend you model all your class relationships in UML class diagrams to get a better understading for them. Anyways, that's it, we are doing domain modeling. We have defined a domain object called Book and a companion BookService object that will handle book operations. Now once you build them and UNIT TEST THEM, yes UNIT TEST THEM (Chapter 23). Then you can use them in your handlers in order to interact with them. As you can see, most of the business rules and logic are encapsulated by these domain objects and not written in your event handlers. This creates a very good design for portability, sustainability and maintainability. So let's start actually seeing how to write all of this instead of imagining it. In Figure 14.4 you can see a more complete class diagram of this simple example.

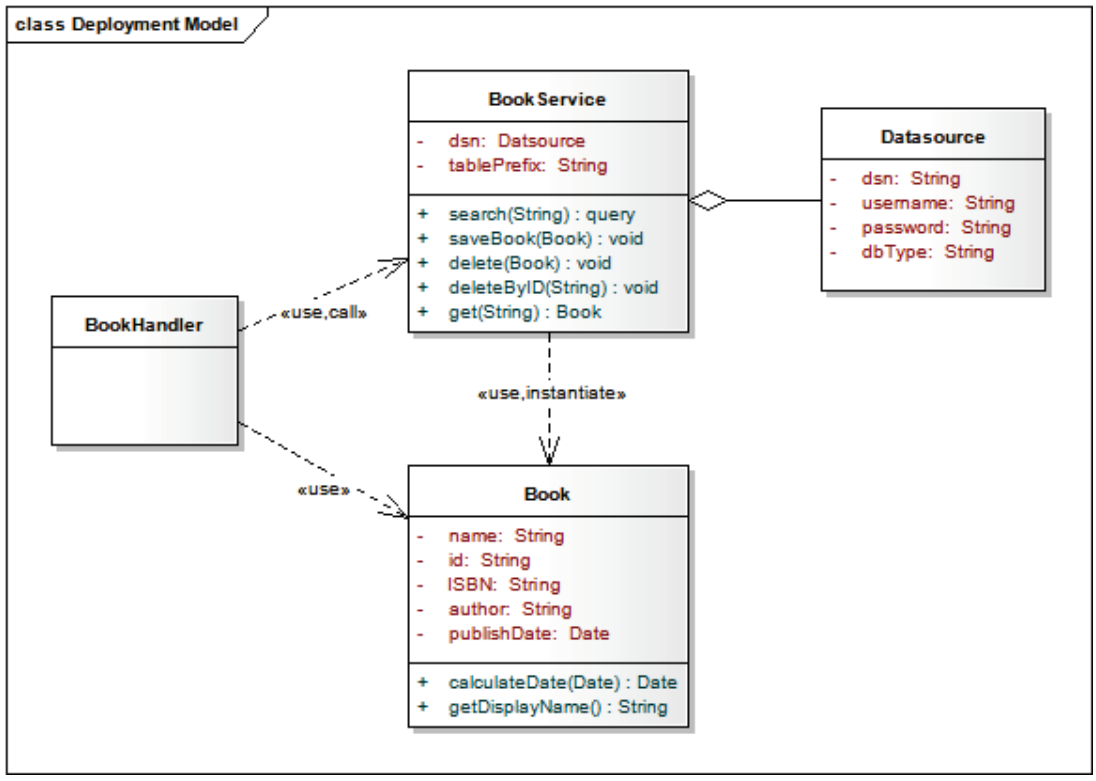


Fig 14.4 : Book Domain Model

Conventions Location

All your model objects will be located in your *model* folder of your application root. This is a convention and can be changed if you so desire by updating the *modelsLocation* setting in the *Conventions* element of your configuration file. You can also change it for the entire framework installation in the ColdBox settings file: *coldbox/config/settings.xml.*

ColdBox Configuration File:

```
<Conventions>
        <handlersLocation></handlersLocation>
        <pluginsLocation></pluginsLocation>
        <layoutsLocation></layoutsLocation>
        <viewsLocation></viewsLocation>
        <eventAction></eventAction>
        <modelsLocation></modelsLocation>
</Conventions>
```

Models External Location

You can also have an external location for your model objects by using a ColdBox setting:

- *ModelsExternalLocation* : This setting is the base instantiation path of the model folder.

```
<Setting name="ModelsExternalLocation" value="coldboxlibrary.models" />
```

This gives you the ability to create centralized locations for model objects that you can easily bring in to your applications. ColdBox 3.0.0 will offer you the ability to have a list of package names that model integration will scan for you.

Important: Model objects in your conventions take precedence over the external location.

Model Configuration Options

There are also several other ColdBox settings that deal with model integration. Below is a nice chart of all the settings you have available.

Setting	Type	Default	Description
ModelsObjectCaching	*boolean*	*true*	Tells the bean factory to cache model objects if cache metadata is found
ModelsSetterInjection	*boolean*	*false*	Use setter injection alongside metadata injection
ModelsDebugMode	*boolean*	*false*	Logs model creation and injections
ModelsDICompleteUDF	*string*	*onDIComplete*	The global name of the UDF to call after injections (if found in CFC)
ModelsStopRecursion	*string (list)*	---	A comma-delimmitted list of class names where the factory should stop recursion Ex: *transfer.com.TransferDecorator*
ModelsExternalLocation	*string*	---	The base instantiation path of where external model objects can be located

```
<Setting name="ModelsObjectCaching" value="true" />
<Setting name="ModelsSetterInjection " value="false" />
<Setting name="ModelsDebugMode" value="false" />
<Setting name="ModelsDICompleteUDF" value="onDIComplete" />
<Setting name="ModelsStopRecursion"
         value="transfer.com.TransferDecorator,model.base.BaseService" />
<Setting name="ModelsExternalLocation" value="externallibrary.models" />
```

Usage Methods

The following usage methods are available in all handlers, plugins and interceptors. The *getModel()* method is also available for all unit testing classes and the ColdBox Proxy.

- **getModel***(string name, [boolean useSetterInjection=false], [string onDICompleteUDF=onDIComplete], [boolean debugMode=false])*
- **populateModel***(any model, [string scope=none], [boolean trustedSetter=false])*

The *getModel()* method has 4 named arguments:

Argument	Required	Default	Description
name	*true*	---	The name or alias of the model object
useSetterInjection	*false*	*false*	You can turn it on to do both setter injection and mixin injection
onDICompleteUDF	*false*	*onDIComplete*	This means that if the object has an *onDIComplete()* method, it will be called after the object has been created and all dependencies have been injected to it
debugMode	*false*	*false*	You can turn it on and it will log out creations and dependencies in your log file
stopRecursion	*false*	---	A comma-delimmitted list of class names that the factory should stop the recursion looking for dependencies on

The *populateModel()* has 3 named arguments:

Argument	Required	Default	Description
model	*true*	---	The name/alias of a model object or an actual instantiated object to populate
scope	*false*	---	If a scope is sent, then the bean factory will populate the variables that match the desired scope name with the request collection name. Great if you do not want to expose setter methods
trustedSetter	*false*	*false*	This flag tells the bean factory to call the setter methods without checking if the setter mehod exists. Great for using implicit setters or onMissingMethod setters

```
<cfset var oUser = getModel('User')>
<cfset populateModel(oUser)>

<!--- OR use the shorthand notation --->
<cfset var oUser = populateModel("User")>
```

Dependencies DSL

ColdBox has a nice DSL (Domain Specific Language) for dependency injection via *cfproperty*, *cffunction* and *cfargument* markers. This is just an extension to what has been available since the ColdBox 2.0X series started. Not only does the DSL type apply to model objects but to anything that is autowired in ColdBox: plugins, handlers, interceptors, ioc produced beans, on demand autowiring and now model objects. Chapter 17 covers extensive autowiring techniques. This section is important because it explains how you can wire up your model objects with the dependencies they need. In addition, it will also show you how to wire up these model objects in your handlers, plugins and interceptors. Below is the *cfproperty* definition:

cfproperty

- **name** : The name of the property to be injected
- **type** : The type of property to inject (see chart)
- **scope** (optional) : Into which scope to inject the object/setting to. Defaults to *variables* scope

The following DSL is how you specify to the framework what dependency you want.

Type	Description
ioc	Get the named ioc bean and inject it. Name comes from the cfproperty name or argument name
ioc:BeanName	Get the ioc bean according to bean name in DSL
ocm	Get the name key from the ColdBox cache and inject it. Name comes from the cfproperty name or argument name
ocm:ObjectKey	Get the object from the ColdBox cache according to DSL object key
model	Get a model with the same name or alias as defined in the cfproperty name="{name}" attribute. Name comes from the cfproperty name or argument name
model:{name}	Same as above but it will get the {name} model object from the DSL and inject it
model:{name}:{method}	Get the {name} model object, call the {method} and inject the results

webservice:{alias}	Get a webservice object using an {alias} that matches in your *coldbox.xml*
coldbox	Get the ColdBox controller
coldbox:setting:{setting}	Get the {setting} setting and inject it
coldbox:plugin:{plugin}	Get the {plugin} plugin and inject it
coldbox:myPlugin:{MyPlugin}	Get the {MyPlugin} custom plugin and inject it
coldbox:datasource:{alias}	Get the datasource bean according to {alias}
coldbox:configBean	Get the config bean object and inject it
coldbox:mailsettingsbean	Get the mail settings bean and inject it
coldbox:loaderService	Get the loader service
coldbox:requestService	Get the request service
coldbox:debuggerService	Get the debugger service
coldbox:pluginService	Get the plugin service
coldbox:handlerService	Get the handler service
coldbox:interceptorService	Get the interceptor service
coldbox:cacheManager	Get the cache manager

Note: The model integration feature supports multiple levels of inheritance. The internal bean factory will inspect all the *cfproperties* and setter methods throughout the inheritance chain.

Constructor & Setter Dependencies

You can easily use the mentioned DSL to wire up a model object's constructor method *init()* by placing a marker annotation on the arguments. A marker annotation is just another attribute to the *cfargument* tag. Remember that ColdFusion allows you to add ANY attributies to certain tags and they will be just considered extra metadata.

For setter methods, you place the marker in the setter method and not the argument. The default attribute is called: *_wireme*. So a simple example would be the following:

```
<!--- Constructor Markers --->
<cffunction name="init" returntype="any" output="false">
  <cfargument name="dsn"      type="any" _wireme="coldbox:datasource:myDSN" />
  <cfargument name="orm" type="transfer.com.Transfer" _wireme="ocm:Transfer" />
</cffunction>

<!--- Setter Markers --->
<cffunction name="setTransfer" type="transfer.com.Transfer" output="true"
            _wireme="ocm:transfer">
</cffunction>
```

As you can see, you use the argument or function marker: *_wireme* to tell the bean factory how to wire up the argument or setter method. What is a bean factory? Well, that is the name of the object that does all these creations and wirings, it is the plugin: BeanFactory. Now, if you do not like the default marker, then you can choose your own. Just create a new setting in your ColdBox.XML.cfm named: *beanfactory_dslMarker*.

```
<Setting name="beanFactory_dslMarker" value="wireit" />

<!--- Then use the wireit marker --->
<cffunction name="init" returntype="any" output="false">
  <cfargument name="dsn" type="any" wireit="coldbox:datasource:myDSN" />
  <cfargument name="transfer" type="transfer.com.Transfer" wireit="ocm" />
</cffunction>
```

Note: ColdBox 3.0.0 will be standardizing on the metadata markers it uses for model integration and autowiring. At the time of writing of this book, the new standard to be used would be the metadata attribute called: **inject**

What Happens If I Don't Put a Metadata Marker?

If you do not place a metadata marker then ColdBox will check to see if you are using an IoC Framework by looking at the *IOCFramework* setting. If the setting is used, then it will default the target type to *ioc*. However, if no IoC Framework has been defined, then the default target type is *model*. This way, if you know that you are injecting model objects or IoC objects, then just ignore the marker and it behaves like ColdSpring setter or constructor injection.

Note: The marker is used only to demarcate using the DSL. If not using the DSL, then it will use the IoC or *model* defaults accordingly.

As you can see from the previous samples, wiring up the constructor argument is fairly easy and very descriptive. You are also not relying on inherited functionality or conflicting code, it is purely metadata that can be ignored if using another factory other than ColdBox.

Injecting Dependencies

You have learned how to wire up the arguments of an object's constructor, now you will learn how to wire up dependencies AFTER the object gets created. So if an object needs dependencies after creation (usually the case), then just use our good old friend *cfproperty* to demarcate or annotate what needs to be injected. The good thing is we just rely on unobtrusive metadata to define what needs to be injected and it can be documented! It can be documented because many of the documentation creation libraries out there can read a component's properties and document them.

```
<!--- Autowire Properties --->
<cfproperty name="myMailSettings"       type="ioc"      scope="instance">

<cfproperty name="ColdBox"              type="coldbox"  scope="instance">

<cfproperty name="ModelsPath"           type="coldbox:setting:ModelsPath"
            scope="instance">

<cfproperty name="Utilities"            type="coldbox:plugin:Utilities"
            scope="instance">

<cfproperty name="ConfigBean"           type="coldbox:configbean"
            scope="instance">

<cfproperty name="MailSettingsBean"     type="coldbox:mailsettingsBean"
            scope="instance">

<cfproperty name="MySiteDSN"            type="coldbox:datasource:mysite"
            scope="instance">

<cfproperty name="testModel"            type="model"
            scope="instance">

<cfproperty name="initDate"             type="model:formBean:getinitDate"
            scope="instance">
```

That is so nice. We can use this DSL to inject almost anything into our model objects. You might be saying that if I have to give my model a name, what is it? Well it is the path from your model directory to your CFC. Again, what if I refactor, I have to change all the references? The answer is no, we have model mappings for that.

Model Mappings

Just by creating a *modelMappings.cfm* in your config folder and calling a simple method from within it, you can create model object aliases. What does this mean? It means you can create an alias name for your object's instantiation path. This will help hide the true class path that can be so essential when refactoring or changing the location of objects. I highly encourage you to do this:

- **addModelMapping***([alias=defaults to the last item in the path],path)*

Argument	Type	Req	Default	Description
alias	*string*	*false*	*last part of the path*	A comma delimmitted list of aliases to match to a specific path.
path	*string*	*true*	---	The instanatiation path of the model object

Remember that the path is the instantiation path from the model folder without the model folder and without '.cfc'. That's it! Just call this method and create alias names for your model objects. What is also much more extensible is that this configuration file is a ColdFusion template, so you can get funky and dynamic. You can do if statements, get data from databases, anything you like.

```
<cfscript>
// Add all the model mappings you want
// addModelMapping(alias="",path="")
addModelMapping('MyFormBean','beans.formBean');

//Adding with a list of aliases
addModelMapping('SecurityService,Security,MySecurity','security.SecurityService'
);
</cfscript>
```

You can also get creative and even dynamically register all your model objects by doing a directory listing and registering all found components. The example above means that you can call the *formBean* object using the alias or the full path:

- **Alias** : *getModel('MyFormBean')*
- **Full Path** : *getModel('beans.formBean')*

Persisting Model Objects

Thanks to metadata and the ColdBox cache, you can use cache metadata attributes and persist your model objects. You can create singletons, transients and even time expired model objects that can adjust to the server's memory demands.

Note: At the time of the writing of this book, ColdBox 3.0.0 was under development and had implemented a shorthand notation for singletons by just saying: *singleton=”true”* in the *cfcomponent* tag instead of what you would see below.

```
<!--- Singleton: Lives for entire application time --->
<cfcomponent name="Model" cache="true" cacheTimeout="0">

<!--- Time Expired Object: Object lives for a max of the default cache object
      timeout in the cache settings --->
<cfcomponent name="Model" cache="true">

<!--- Time Expired Object: Object lives for a max of 30 minutes --->
<cfcomponent name="Model" cache="true" cacheTimeout="30">

<!--- Time Expired Object: Object lives for a max of 40 minutes, but if not used
      for the past 15 minutes expire it --->
<cfcomponent name="Model" cache="true" cacheTimeout="40"
cacheLastAccessTimeout="15">

<!--- Transient: Used on demand --->
<cfcomponent name="Model">
```

You can create incredible cache sensitive models, just by tapping into the ColdBox cache. What is also an added benefit is that all model object's metadata are internally cached in a metadata dictionary. So creating model objects, even transients, are FAST!

Important: Please remember that the ColdBox cache is based on a solid memory sensitive cache. So objects that have timeouts are not guaranteed to live the entire length of the timeout because the JVM can request memory and purge them for you. If this happens, the framework will re-cache the objects a second time seamlessly for you. You do not have to worry about their persistence. It is all done for you.

Simple Example

This section shows a simple user service, gateway, user object and how to use them within a handler. Below is a diagram of our model folder layout.

```
+ handlers
   + user.cfc
+ model
  + security
     + UserService.cfc
     + UserGateway.cfc
     + User.cfc
```

Coldbox.XML

This is a snippet of the configuration file:

```
<Datasources>
  <Datasource alias="dsn" name="MySite" dbtype="mysql" />
</Datasources>
```

Model Mappings

Some mappings are created so the CFCs can be referenced by an alias and not their class path:

```
//No alias is used, the alias will be the last part of the path.
addModelMapping(path='security.UserService');
addModelMapping(path='security.UserGateway');
```

User Service

This user service is a simple CFC that just has a gateway dependency for complex queries:

```
<!--- Cache of 0 = singleton --->
<cfcomponent name="UserService" output="true" cache="true" cacheTimeout="0">

<!--- Dependencies --->
<cfproperty name="UserGateway" type="model" scope="instance" />
<cfproperty name="SessionStorage" type="coldbox:plugin:sessionstorage"
            scope="instance" />

<cfscript>
instance = structnew();
</cfscript>

<cffunction name="init" output="false" returntype="UserService">
   <cfreturn this>
</cffunction>

<cffunction name="getAllUsers" output="false" access="public" returntype="query"
           hint="Returns all users in the database, active and inactive.">

<cfargument name="orderProperty"        type="string"  required="false"
            default=""/>
<cfargument name="orderASC"             type="boolean" required="false"
           default="true" hint="Order ASC = true, DESC = false"/>
```

```
<cfscript>
var query = "";
query =
instance.UserGateway.findUsers(arguments.orderProperty,arguments.orderASC);

return query;
</cfscript>
</cffunction>

<cffunction name="authenticateUser" output="false" access="public"
            returntype="boolean" hint="Authenticate a User. If valid it places
            them in session. Returns true if user is valid and authenticated and
            ready for usage.">

<cfargument name="username" type="string" required="true"/>
<cfargument name="password" type="string" required="true"/>

<cfscript>
// Prepare results
var authenticated = false;
var oUser = "";

// Try to get user by credentials
oUser = getUserByCredentials(argumentCollection=arguments);

//Is User in system.
if ( oUser.getIsPersisted() ){
  //Save User State
  instance.sessionstorage.setVar('CurrentUser', oUser);
  //Set Return Flags
  authenticated = true;
}

return authenticated;
</cfscript>
</cffunction>

<!--- Get User By Credentials --->
<cffunction name="getUserByCredentials" output="false" access="public"
            returntype="User" hint="Returns an active/confirmed user by its
            credentials">

<cfargument name="username" type="string" required="true"/>
<cfargument name="password" type="string" required="true" hint="This argument is
            hashed internally."/>

<cfscript>
var oUser = "";
var sqlProps = structnew();
```

```
// prepare sqlProps
sqlProps.username = arguments.username;
sqlProps.password = hash(arguments.password,'SHA-512');
sqlProps.isConfirmed = 1;
sqlProps.isActive = 1;

// Create User
oUser = createObject("component","User").init();

// Try to get user now.
instance.userGateway.readByProperties(oUser,sqlProps);

return oUser;
</cfscript>
</cffunction>

<!--- Get A User Session --->
<cffunction name="getUserSession" output="false" access="public"
            returntype="User" hint="This method checks if a user is in an
            authorized session, else it returns the default user object.">
<cfscript>
var oUser = "";

//Is user in session
if ( instance.sessionstorage.exists( 'CurrentUser' ) ){
  oUser = instance.sessionstorage.getVar( 'CurrentUser' );
}
else{
  oUser = createObject("component","User");
}

return oUser;
</cfscript>
</cffunction>

<!--- Clean a user's session. --->
<cffunction name="cleanUserSession" output="false" access="public"
            returntype="void" hint="This method will clean the user session.">
<cfscript>
instance.sessionstorage.deleteVar( 'CurrentUser' );
</cfscript>
</cffunction>

</cfcomponent>
```

User Gateway

This user gateway is a simple CFC that does complex queries on the database for user operations. I separated it into a gateway object, because I plan to have lots and lots of complex queries for users. If you where doing simple queries or an ORM, maybe just having a service layer would suffice. Again, don't think

that everything needs a service-gateway combination and especially 1-1 relationships between tables and objects. Remember that objects must have identity and service layers can manage several tables as long as they provide cohesion and have well laid out responsibilities.

```
<!--- I will just lay out one method not all --->
<!--- Cache of 0 = singleton --->
<cfcomponent name="UserGateway" output="true" cache="true" cacheTimeout="0">

<!--- Dependencies --->
<cfproperty name="dsn" type="coldbox:datasource:dsn" scope="instance" />

<cfscript>
instance = structnew();
</cfscript>

<cffunction name="init" output="false" returntype="UserService">
   <cfreturn this>
</cffunction>

<cffunction name="getAllUsers" output="false" access="public" returntype="query"
            hint="Returns all users in the database, active and inactive.">

<cfargument name="orderProperty"        type="string"  required="false"
            default=""/>
<cfargument name="orderASC"             type="boolean" required="false"
           default="true" hint="Order ASC = true, DESC = false"/>

<cfset var qUser = 0>

<cfquery name="qUser" datasource="#instance.dsn.getName()#">
select * from users
order by #arguments.orderProperty# #arguments.orderASC#
</cfquery>

<cfreturn qUser>

</cffunction>

</cfcomponent>
```

Handler Code

This is some handler code for a *user* handler.

```
<cfcomponent name="User" output="false" extends="coldbox.system.eventhandler"
             autowire="true">

<!--- Dependencies --->
<cfproperty name="UserService" type="Model" scope="instance" />

<cffunction name="list" output="false" returntype="void">
<cfargument name="event" type="any">
<cfscript>
var rc = event.getCollection();

//get all users
rc.qUsers = instance.UserService.getAllUsers();

//View
event.setView('users/list');
</cfscript>
</cffunction>

<cffunction name="login" output="false" returntype="void">
<cfargument name="event" type="any">
<cfscript>
event.setView("user/login");
</cfscript>
</cffunction>

<cffunction name="doLogin" output="false" returntype="void">
<cfargument name="event" type="any">
<cfscript>
//Authenticate
if( instance.UserService.authenticate(event.getValue("username",""),
                                      event.getValue("password","")) ){
  setNextEvent('user.home');
}
else{
  getPlugin("messagebox").setMessage("warning",
                          "Username and password not valid. Please try again");
  setNextEvent('user.login');
}
</cfscript>
</cffunction>
```

```
<cffunction name="doLogout" output="false" returntype="void">
<cfargument name="event" type="any">
<cfscript>
instance.UserService.cleanUserSession();
setNextEvent('user.login');
</cfscript>
</cffunction>

</cfcomponent>
```

Summary

The ColdBox model architecture leverages conventions, caching and a new dependency injection mechanisms by using DSL markers that will take your development to new RAD (Rapid Application Development) heights. Just by the fact that you can leverage conventions simplifies the wiring and creation of model objects. Also, by having a rock solid caching engine behind your domain model objects, allows you more granular control over objects that can make wise usage of system resources. Model integration is still in its infancy and future versions of ColdBox will continue to break barriers and push this approach to new heights. Enjoy and start building great domain models!