

[<< Back to Dashboard](#)

Model Integration Guide

Covers up to version 3.5.0

Introduction



Model integration helps you create, manage, and use model (business logic) objects very easily within your ColdBox application via [WireBox](#). WireBox, is our dependency injection and AOP framework, that will do all the magic of building, wiring objects with dependencies and helping your persist objects in some state (singletons, transients, request, etc). The main purpose for model integration is to make developer's development workflow easier! And we all like that Easy button!

This integration will give you a good kick start on dependency injection, caching, persistence, etc without you actually studying for it. 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()`
- Easily handle model or handler dependencies by using `cfproperty` and constructor argument conventions.
- A conventions DSL (Domain Specific Language) has been created to facilitate what needs to be injected in the models (Don't shiver with fear yet, please keep reading)
- Easily determine what scope model objects should be persisted in: Transients, Singletons, Cache, Request, Session, etc.
- Easily create a configuration binder to create aliases or complex object relationships (Java, WebServices, RSS, etc.)
- Easily populate model objects with data from a request: `populateModel()`
- Easily `validate` model objects using our awesome `ValidBox` validation engine and `validateModel()`
- Integrate with ColdFusion ORM via our `ActiveEntity` class if you like Active Record pattern or
- Integrate with ColdFusion ORM via our `ORM Service` or `Virtual Entity Services`.

Wow! You can do all that? Yes and much more. So let's begin.

Domain Modeling

The Model layer represents your data structures and business logic. 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.

UML Resources: There are tons of great UML resources and tools. Here are some great tools for you:

- Sparx Systems Enterprise Architect - <http://www.sparxsystems.com/products/ea/index.html>
- ColdFusion generation templates for Enterprise Architect - <https://github.com/Imajano/EA-ColdFusion-CodeGeneration>
- ArgoUML - <http://argouml.tigris.org/>
- Poseidon UML - <http://www.gentleware.com/>
- Learning UML - <http://shop.oreilly.com/product/9780596009823.do>
- UML in a nutshell - http://shop.oreilly.com/product/9780596007959.do?green=87D3081D-5A0D-50F7-9757-95B7E8779516&cmp=af-mybuy-9780596007959_IP

Let's say that I want to build a simple book catalog and I want to be able to do the following:

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

There are several ways I can go about this and your design will depend on the tools you use. If you use an ORM like ColdFusion ORM you most likely will use either an [ActiveEntity](#) approach or build [virtual service layers](#) or build a service layer based on our [Base ORM services](#). If you are not using an ORM, then most likely you will have to build all object, service and data layers manually. We will concentrate first on the last approach first to do our modeling and then build them out in different ways.

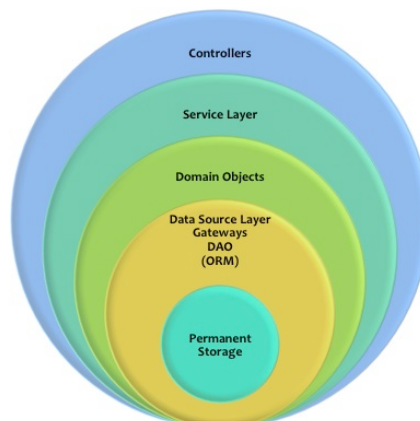
Service Layer

I want to apply best practices and use a service layer approach for my application and model design. I will then use these service objects in my handlers in order to do the business logic for me. 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` - A service layer for book operations
- `Book.cfc` - Represents a book in my system



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 coor-dinating 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. There are basically two approaches when

Contents

- [Model Integration Guide](#)
 - [Introduction](#)
 - [Domain Modeling](#)
 - [Service Layer](#)
 - [Data Layers](#)
 - [Book](#)
 - [Book Service](#)
 - [Conventions Location](#)
 - [WireBox Binder](#)
 - [Super Type Usage Methods](#)
 - [getModel\(\)](#)
 - [populateModel\(\)](#)
 - [validateModel\(\)](#)
 - [Injection DSL](#)
 - [Model Object Namespace](#)
 - [EntityService Namespace](#)
 - [ColdBox Namespace](#)
 - [Object Scopes](#)
 - [Mapping Objects](#)
 - [Coding: Solo Style](#)
 - [Datasource](#)
 - [Contact.cfc](#)
 - [ContactDAO.cfc](#)
 - [ContactService.cfc](#)
 - [Contacts_Handler](#)
 - [Summary](#)
 - [Coding: ActiveEntity Style](#)
 - [ORM](#)
 - [Contact.cfc](#)
 - [Contacts_Handler](#)
 - [Views](#)
 - [index.cfm](#)
 - [editor.cfm](#)
 - [Summary](#)
 - [Coding: Virtual Service Layer Style](#)
 - [ORM](#)
 - [Contact.cfc](#)
 - [Contacts_Handler](#)
 - [Views](#)
 - [index.cfm](#)
 - [editor.cfm](#)
 - [Summary](#)

building a service layer:

1. Designing the services around functionality that might encompass several domain model objects. This is our preferred approach as it creates a more rich service layer and you will not have class explosion.
2. Designing a service for each business object, which in turn matches a table or set of tables in the permanent storage. This approach is easy to follow, but the consequences are the responsibilities that could be grouped are not and you will end up with class explosion.

The best way to determine what you prefer or need is to actually try both approaches. Once you design them and put them in practice, you will find your preference. 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, the art of knowing when your code is not right and therefore smells nasty!

The **BookService** object will be my API to do operations as 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 the **BookService** object and will be used by event handlers in order to populate and validate them with data from the user. The view layer will also use the **Book** object in order to present the data. As you can see, 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 book service for persistence.

Data Layers

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 data gateway object. 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 (Encompasses 1 or more persistence tables) and create gateways or data layers when needed, especially when not using an ORM. 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 make you 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.

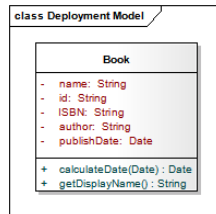
Book

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

- name
- id
- createdata
- 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. 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



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!

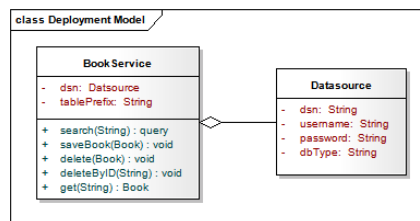
Book Service

Back to the book service 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 stuff. It might also need a table prefix to use (because I want to) and it comes from a setting in my application's configuration file. Ok, so now we know the following dependencies or external forces:

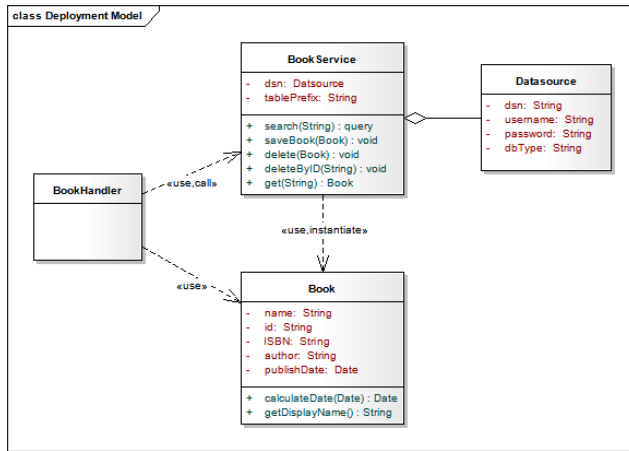
- A datasource (as an object or string)
- A setting (as a string)

I can also now think of a few methods that I can have on my book service:

- `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)` Save or Update a book
- `deleteBook(book:Book)` Delete a book



I recommend you model these class relationships in [UML class diagrams](#) to get a better feeling of it. 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. 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. Below you can see a more complete class diagram of this simple example.



So now that you exercised your mind and modeled your problem, you are ready to start coding and learning how to put this baby together!

Conventions Location

All your model objects will be located under your **model** folder of your application root by convention (See [Conventions](#)).

WireBox Binder

You can have an optional [WireBox](#) configuration binder that can fine-tune the WireBox engine and also where you can create fancy object mappings, aliases and even more model locations by convention. Usually you will find this binder by convention in your `config/WireBox.cfc` location and it looks like this:

```

component extends= "coldbox.system.ioc.config.Binder" {
    function configure(){
        // Configure WireBox
        wireBox = {
            // Scope registration, automatically register a wirebox injector instance on any CF scope
            // By default it registers itself on application scope
            scopeRegistration = {
                enabled = true,
                scope = "application" , // server, cluster, session, application
                key = "wireBox"
            },

            // Custom DSL Namespace registrations
            customDSL = {
                // namespace = "mapping name"
            },

            // Custom Storage Scopes
            customScopes = {
                // annotationName = "mapping name"
            },

            // Package scan locations or model external locations by convention
            scanLocations = [],

            // Stop Recursions
            stopRecursions = [],

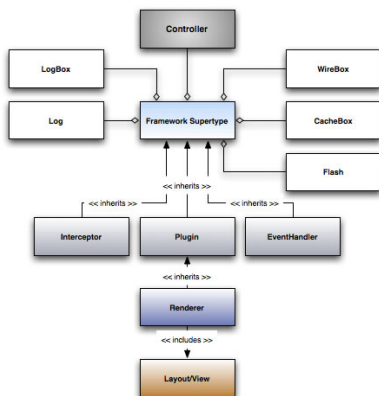
            // Parent Injector to assign to the configured injector, this must be an object reference
            parentInjector = "",

            // Register all event listeners here, they are created in the specified order
            listeners = [
                // { class="", name="", properties={} }
            ];
        };

        // Map Bindings below
    }
}
    
```

Super Type Usage Methods

ColdBox Major Classes



The following usage methods are available in all handlers, plugins and interceptors thanks to our Framework Super Type and will be used so you can retrieve, populate and validate model objects.

- `getModel([name=""], [dsl=""], [initArguments={}])`
- `populateModel(any model, [scope=none], [boolean trustedSetter=false],[string include=],[string exclude=])`
- `validateModel(target, [fields=*], constraints="", [locale=""])`

getModel()

Argument	Require	Default	Description
name	false	---	The name/alias or full instantiation path of a model object to retrieve from WireBox.
dsl	false	---	The DSL injection string to retrieve an object by instead of using a name/alias or instantiation path. (See WireBox Injection DSL)
initArguments	false	{}	An optional structure of arguments to initialize the object with

```
// Retrieve the User.cfc in the model folder
var oUser = getModel( 'User' );
// Retrieve the User.cfc in the model/users folder
var oUser = getModel( "users.User" );
// Retrieve the User using an alias you mapped in your configuration binder
var oUser = getModel( "MyUser" );
// Retrieve an object using a full instantiation path
var oUtil = getModel( "mypath.utilities.MyUtil" );
```

populateModel()

ColdBox can populate model objects from data in the request collection by matching the name of the form element to the name of a property on the object. You can also populate model objects from JSON, XML, Queries and other structures a-la-carte by talking directly to [WireBox's](#) object populator.

Argument	Require	Default	Description
model	true	---	The name/alias or instantiation path of a model object or an actual instantiated object to populate.
scope	false	---	If a scope is sent, then WireBox will populate the variables that match the desired scope name with the request collection name. In other words it injects the values into the appropriate scope you chose instead of calling setter methods.
trustedSetter	false	false	Defaults to false, this flag tells WireBox to call the setter methods without checking if the setter method exists. Great for using implicit setters or onMissingMethod setters.
include	false		A list of keys to include from the public request collection when populating.
exclude	false		A list of keys to exclude from the public request collection when populating.

```
var oUser = getModel( 'User' );
populateModel(oUser);

// or
var oUser = populateModel( "User" );
// populate and exclude the primary key
var oUser = populateModel(model= "User",exclude= "userID" );
```

validateModel()

Uses [ValidBox](#), our very own form and object validation engine, or any third party connected validation engine. The results of the validation call is a validation results object that must adhere to our validation interface: `coldbox.system.validation.result.IValidationResult` and most likely it will be: `coldbox.system.validation.result.ValidationResult`. By now you should now that the best way to find out about the methods in these objects is to look for them in our [API Docs](#).

Argument	Require	Default	Description
target	true	---	The instantiated object to validate or a structure (RC/PRC) of data to validate.
fields	false	*	Validate on all or one or a list of fields (properties) on the target, by default we validate all fields declared in its constraints
constraints	false	---	The shared constraint name to use, or an actual constraints structure or none at all so it will discover the constraints on the object itself.
locale	false	---	The locale to validate in, meaning all the validation messages comes from resource bundles the framework manages.

I know we have not seen how to validate yet, we will shortly, but also take a peek at our [Validation](#) docs as well.

```
var user = populateModel( entityNew( "User" ) );
var vResults = validateModel( user );
if ( vResults.hasErrors() ){
    getPlugin( "MessageBox" ).error(messageArray=vResults.getAllErrors());
    setNextEvent( "users.edit" );
}
else {
    userService.save( user );
    setNextEvent( "users.list" );
}
```

Injection DSL

Before we start building our objects we need to know how to tell WireBox to inject objects for us. You can do all this via configuration (Inside the binder) like any other DI framework, but the easiest approach is to use our injection annotation and conventions. The injection DSL can be applied to `property`, `cargument`, `cfunction` or called via `getModel()`. It is represented by adding an attribute called `inject`. This attribute is like your code is shouting, *Hey, I need something here, hello, I need something!*. That something can be another object, setting, cache, etc.

```
// property injection
property name= "service" inject= "MyService" ;

// setter injection
function setMyService(MyService) inject= "MyService" {
    variables.MyService = arguments.MyService;
}

// argument injection
<cargument name= "setting" inject= "coldbox:setting:MyDSN" >
/**
```

```
* @myDAO.inject model:MyDAO
*/
function init(myDAO){
}
}
```

The value of the **inject** attribute is what we call our injection DSL. This string represents the concept of retrieving an object WireBox knows about, which can be an object, a setting, a datasource, your custom data, etc. You can see all the different Injection DSL's in the WireBox documentation: (See [Injection DSL](#)). Below are just the most common ones we will use:

Model Object Namespace

The default namespace is not specifying one. This namespace is used to retrieve either named mappings or full component paths.

DSL	Description
empty	Same as saying <i>id</i> . Get a mapped instance with the same name as defined in the property, argument or setter method.
id	Get a mapped instance with the same name as defined in the property, argument or setter method.
id:{name}	Get a mapped instance by using the second part of the DSL as the mapping name.
id:{name}:{method}	Get the {name} instance object, call the {method} and inject the results
model	Get a mapped instance with the same name as defined in the property, argument or setter method.
model:{name}	Get a mapped instance by using the second part of the DSL as the mapping name.
model:{name}:{method}	Get the {name} instance object, call the {method} and inject the results

```
// Let's assume we have mapped a few objects called: UserService, SecurityService and RoleService
// Empty inject, use the property name, argument name or setter name
property name= "userService" inject;
// Using the name of the mapping as the value of the inject
property name= "security" inject= "SecurityService" ;
// Using the full namespace
property name= "userService" inject= "id:UserService" ;
property name= "userService" inject= "model:UserService" ;
// Simple factory method
property name= "roles" inject= "id:RoleService:getRoles" ;
```

EntityService Namespace

Gives you the ability to easily inject base orm services or binded virtual entity services for you:

DSL	Description
entityService	Inject a BaseORMService object for usage as a generic service layer
entityService:{entity}	Inject a VirtualEntityService object for usage as a service layer based off the name of the entity passed in.

```
// Generic ORM service layer
property name= "genericService" inject= "entityService" ;
// Virtual service layer based on the User entity
property name= "userService" inject= "entityService:User" ;
```

ColdBox Namespace

This namespace is a combination of namespaces that are only active when used within a ColdBox application:

DSL	Description
coldbox	Get the coldbox controller reference
coldbox:flash	Get a reference to the application's flash scope object
coldbox:setting:{setting}	Get the coldbox application {setting} setting and inject it
coldbox:setting:{setting}@{module}	Get the coldbox application {setting} from the {module} 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:myPlugin:{MyPlugin}@{module}	Get the {MyPlugin} custom plugin from the {module} module and inject it
coldbox:datasource:{alias}	Get a new datasource bean according to {alias}
coldbox:configBean	Get a new config bean object and inject it
coldbox:mailSettingsBean	Get a new mail settings bean and inject it
coldbox:loaderService	Get a reference to the loader service
coldbox:requestService	Get a reference to the request service
coldbox:debuggerService	Get a reference to the debugger service
coldbox:pluginService	Get a reference to the plugin service
coldbox:handlerService	Get a reference to the handler service
coldbox:interceptorService	Get a reference to the interceptor service
coldbox:moduleService	Get a reference to the ColdBox Module Service
coldbox:interceptor:{name}	Get a reference of a named interceptor {name}
coldbox:cacheManager	get the cache manager
coldbox:fwConfigBean	Get a configuration bean object with ColdBox settings instead of Application settings
coldbox:fwSetting:{setting}	Get a setting from the ColdBox settings instead of the Application settings
coldbox:moduleSettings:{module}	Inject the entire {module} settings structure
coldbox:moduleConfig:{module}	Inject the entire {module} configurations structure
ioc	Get the named ioc bean and inject it. Name comes from the cfproperty, setter or argument name
ioc:{beanName}	Get the ioc bean according to {beanName}
javaLoader:{class}	Create an object from the JavaLoader plugin and its set of loaded java libraries
webservice:{alias}	Get a webservice object using an {alias} that matches in your coldbox configuration file.

```
// some examples
property name= "logbox" inject= "logbox" ;
property name= "rootLogger" inject= "logbox:root" ;
property name= "logger" inject= "logbox:logger:model.com.UserService" ;
property name= "moduleService" inject= "coldbox:moduleService" ;
property name= "producer" inject= "coldbox:interceptor:MessageProducer" ;
property name= "configBean" inject= "coldbox:fwConfigBean" ;
property name= "producer" inject= "interceptor:MessageProducer" ;
property name= "appPath" inject= "coldbox:fwSetting:ApplicationPath" ;

// JavaLoader goodness
property name= "binaryHeap" inject= "javaLoader:org.apache.commons.collections.BinaryHeap" ;
property name= "email" inject= "javaLoader:org.apache.commons.mail.SimpleEmail" ;
```

Object Scopes

You can very easily add persistence to your domain objects via our annotations or binder configuration. This is of great benefit as you can control where these objects will be scoped in the ColdFusion engine. The available scopes are:

- **transient or no scope** : The default scope. Meaning objects have no scope, they are recreated every single time you request them.
- **singleton** : Objects are created once and live until your Application expires
- **cachebox** : You can store your objects in any [CacheBox](#) provider and even provide timeouts for them
- **session** : Store them in the ColdFusion session scope
- **server** : Store them in the ColdFusion server scope
- **request** : Store them in the ColdFusion request scope
- **application** : Store them in the ColdFusion application scope
- **CUSTOM** : You can build your own scopes as well.

```
// transient
component name= "MyService" {}

// singleton
component name= "MyService" singleton{}

// cache in default provider
component name= "MyService" cache= "true" cacheTimeout= "45" cacheLastAccessTimeout= "15" {}

// cache in another provider
component name= "MyService" cachebox= "MyProvider" cacheTimeout= "45" {}

// request scope
component name= "MyService" scope= "request" {}

// session
component name= "MyService" scope= "session" {}
```

Please note that using annotations is optional, you can configure every object in our configuration binder as well.

Mapping Objects

You can use the [WireBox](#) configuration binder to map object relationships, aliases, etc. By convention you retrieve objects that exist in the **model** folder with their appropriate name and path. This is great and dandy, but sometimes if we need to refactor our paths, most likely our code will have to change. If you want to prepare for the future and be extensible, the best approach is to create aliases for your objects so you retrieve them by alias instead of path locations. You can do this using our [mapping DSL](#).

```
// Create an alias of MyService for the full instantiation path
map( "MyService" ).to( "#appmapping#.model.users.MyService" );

// Map the entire model directory and the alias is the name of the CFC
mapDirectory( "#appMapping#.model" );
```

Important: The variable `appMapping` is the location of your application in the web root. Always use it so your application is portable.

Coding: Solo Style

Now that we have seen all the theory and stuff, let's get down to business and do some examples. We will start with the full coding approach with no ORM and then spice it up with ORM, so you can see how awesome ORM can be. The examples will not show the entire application being built, but enough to get you started with the process of modeling everything. Here is a layout of what we will build:

```
+ handlers
+ contacts.cfc
+ model
+ ContactService.cfc
+ ContactDAO.cfc
+ Contact.cfc
```

I will create a DAO for this small example, so we can showcase how to talk to multiple objects.

Datasource

Make sure you register a datasource in your ColdFusion administrator, we called ours **contacts** and then register it in your ColdBox configuration so ColdBox can build datasource objects for us. This is totally optional, but we do it to showcase more injections and dealing with more objects.

```
datasources = {
contacts = {name= "contacts" , dbtype= "mysql" }
};
```

Contact.cfc

An object that represents a contact and self-validates using [ColdBox Validation](#):

```
component accessors= "true" {

// properties
property name= "firstName" ;
property name= "lastName" ;
property name= "email" ;

// validation
this.constraints = {
firstName = {required= true},
lastName = {required= true},
email = {required= true , type= "email" }
};

function init(){
return this ;
}
```

}

ContactDAO.cfc

Our Contact DAO will talk to the datasource object we declared and do a few queries. Notice that this object is a singleton and has some dependency injection.

```
component accessors= "true" singleton{
    // Dependency Injection
    property name= "dsn" inject= "coldbox:datasource:contacts" ;

    function init(){
        return this;
    }

    query function getAll(){
        var q = new Query(datasource= "#dsn.getName()#" ,sql= "SELECT * FROM contacts" );
        return q.execute().getResult();
    }

    query function getContact(required contactID){
        var q = new Query(datasource= "#dsn.getName()#" ,sql= "SELECT * FROM contacts where contactID = :contactID" );
        q.addParam(name= "contactID" ,value=arguments.userID,cfsqltype= "cf_sql_numeric" );
        return q.execute().getResult();
    }

    ... ALL OTHER METHODS HERE FOR CRUD ....
}
```

ContactService.cfc

Here is our service layer and we have added some logging just for fun :). Notice that this object is a singleton and has some dependency injection.

```
component accessors= "true" {
    // Dependency Injection
    property name= "dao" inject= "ContactDAO" ;
    property name= "log" inject= "logbox:logger:{this}" ;
    property name= "populator" inject= "wirebox:populator" ;
    property name= "wirebox" inject= "wirebox" ;

    function init(){
        return this;
    }

    /**
     * Get all contacts as an array of objects or query
     */
    function list(boolean asQuery= false){
        var q = dao.getAllUsers();
        log.info( "Retrieved all contacts" , q.recordcount);

        if( asQuery ){ return q; }

        // convert to objects
        var contacts = [];
        for( var x=1; x lte q.recordcount; x++){
            arrayAppend( contacts, populator.populateFromQuery( wirebox.getInstance( "Contact" ), q, x ) );
        }

        return contacts;
    }

    /**
     * Get a persisted contact by ID or new one if 0 or no records
     */
    function get(required contactID=0){
        var q = dao.getContact(arguments.contactID);
        // if 0 or no records
        if( contactID eq 0 OR q.recordcount eq 0 ){
            // return a new object
            return wirebox.getInstance( "Contact" );
        }
        // Else return the object
        return populator.populateFromQuery( wirebox.getInstance( "Contact" ), q, 1 );
    }

    ... ALL OTHER METHODS HERE ....
}
```

Now, some observations of the code:

- We use the **populator** object that is included in [WireBox](#) to make our lives easier so we can populate objects from queries and deal with objects.
- We also inject a reference to the object factory [WireBox](#) so it can create **Contact** objects for us. Why? Well what if those objects had dependencies as well.

Contacts Handler

Let's put all the layers together and make the handler talk to the model. I create different saving approaches, to showcase different integration techniques:

```
component{
    // Dependency Injection
    property name= "contactService" inject= "ContactService" ;

    function index(event,rc,prc){
        // Get all contacts
        prc.aContacts = contactService.list();
        event.setView( "contacts/index" );
    }

    function newContact(event,rc,prc){
        event.setView( "contacts/newContact" );
    }

    function create(event,rc,prc){
        var contact = populateModel( "Contact" );
        // validate it
        var vResults = validateModel(contact);
        // Check it
        if( vResults.hasErrors() ){
            getPlugin( "MessageBox" ).error(messageArray=vResults.getAllErrors());
            return newContact(event,rc,prc);
        }
        else {

```

```

    getPlugin( "MessageBox" ).info( "Contact created!" );
    setNextEvent( "contacts" );
}
}

function save(event,rc,prc){
event.paramValue( "contactID" ,0);
var contact = populateModel( contactService.get( rc.contactID ) );
// validate it
var vResults = validateModel(contact);
// Check it
if( vResults.hasErrors() ){
getPlugin( "MessageBox" ).error(messageArray=vResults.getAllErrors());
return newContact(event,rc,prc);
}
else {
getPlugin( "MessageBox" ).info( "Contact created!" );
setNextEvent( "contacts" );
}
}
}
}

```

Summary

We have now put all our concepts together and created something that talks to all layers. We do not show the view code, but we leave that as homework :). However, this approach is great, but we can do better by leveraging ColdFusion ORM.

Coding: ActiveEntity Style

Now let's build the same thing but using ColdFusion ORM and our [ORM:ActiveEntity:ActiveEntity](#) approach:

```

+ handlers
+ contacts.cfc
+ model
+ Contact.cfc

```

ORM

You will first make sure your **contacts** datasource exists in the Administrator and then we can declare our ORM settings in our **Application.cfc**

```

// ORM Settings
this.ormEnabled = true;
this.datasource = "contacts";
this.ormSettings = {
cfclocation = "model",
dbcreate = "update",
logSQL = true,
flushAtRequestEnd = false,
autoManageSession = false,
eventHandling = true,
eventHandler = "coldbox.system.orm.hibernate.WBEventHandler"
};

```

```
ORMReload();
```

These are the vanilla settings for using the ORM with ColdBox. Make sure that **flushAtRequestEnd** and **autoManageSession** are set to **false** as ColdBox will manage that for you. In this example, we also use **dbcreate="update"** as we want ColdFusion ORM to build the database for us which allows us to concentrate on the domain problem at hand and not persistence. Finally, add an **ormReload()** at the end for now as we are in development mode and want ORM changes to take effect immediately. For production, remove it. You also see that we add our own **eventHandler** which points to the vanilla [WireBox](#) event handler so we can make Active Entity become well, Active!

Now open your **ColdBox.cfc** and add the following to activate ORM injections:

```
orm = { injection = {enabled= true } };
```

Contact.cfc

An object that represents a contact and self-validates using [ColdBox Validation](#), and is an awesome [ORM:ActiveEntity](#):

```

/**
 * A cool Contact entity
 */
component persistent= "true" table= "contacts" extends= "coldbox.system.orm.hibernate.ActiveEntity" {

// Primary Key
property name= "contactID" fieldtype= "id" column= "contactID" generator= "native" setter= "false";

// Properties
property name= "firstName" ormtype= "string";
property name= "lastName" ormtype= "string";
property name= "email" ormtype= "string";

// validation
this.constraints = {
firstName = {required= true},
lastName = {required= true},
email = {required= true, type= "email" }
};
}
}

```

Contacts Handler

That's right, go to the handler now, no need of data layers or services, we build them for you! This time, we show you the entire CRUD operations as Active Entity makes life easy!

```

/**
 * I am a new handler
 */
component{

function index(event,rc,prc){
prc.contacts = entityNew( "Contact" ).list(sortOrder= "lastName" ,asQuery= false);
event.setView( "contacts/index" );
}

function editor(event,rc,prc){
event.paramValue( "id" ,0);
prc.contact = entityNew( "Contact" ).get( rc.id );
event.setView( "contacts/editor" );
}
}

```

```

function delete(event,rc,prc){
    event.paramValue( "id",0);
    entityNew( "Contact" ).deleteByID( rc.id );
    getPlugin( "MessageBox" ).info( "Contact Removed!" );
    setNextEvent( "contacts" );
}

function save(event,rc,prc){
    event.paramValue( "id",0);
    var contact = populateModel( entityNew( "Contact" ).get( rc.id ) );
    if( contact.isValid() ){
        contact.save();
        getPlugin( "MessageBox" ).info( "Contact Saved!" );
        setNextEvent( "contacts" );
    }
    else {
        getPlugin( "MessageBox" ).error(messageArray=contact.getValidationResults().getAllErrors());
        return editor(event,rc,prc);
    }
}
}

```

Views

Here are the views as well:

index.cfm

```

<<cfoutput>
<h1>Contacts </h1>
#getPlugin( "MessageBox" ).renderit()#
#html.href(href= 'contacts.editor' ,text= "Create Contact" )#
<br><br>
<cfloop array= "#prc.contacts#" index= "contact" >
<div>
#contact.getLastName()# , #contact.getFirstName()# (#contact.getEmail()#) <br />
#html.href(href= 'contacts.editor.id.#contact.getContactID()#' ,text= "[ Edit ]" )#
#html.href(href= 'contacts.delete.id.#contact.getContactID()#' ,text= "[ Delete ]" ,onclick= "return confirm('Really Delete?')" )#
<hr>
</div>
</cfloop>
</cfoutput>

```

editor.cfm

Check out our awesome `html` helper object. It can even build the entire forms according to the Active Entity object and bind the values for you!

```

<<cfoutput>
<h1>Contact Editor </h1>
#getPlugin( "MessageBox" ).renderit()#
#html.startForm(action= "contacts.save" )#
#html.entityFields(entity=prc.contact,fieldwrapper= "div" )#
#html.submitButton()# or #html.href(href= "contacts" ,text= "Cancel" )#
#html.endForm()#
</cfoutput>

```

Summary

The ColdBox [Active Entity](#) object really makes life easy! It can give us all kinds of CRUD operations, searching, paging, and so much more.

Coding: Virtual Service Layer Style

Now let's build the same thing but using ColdFusion ORM and our [Virtual Service Layer](#) approach, in which we will use a service layer but virtually built by ColdBox. This will most likely give you 80% of what you would ever need, but in case you need to create your own and customize, then you would build a service object that extends or virtual or base layer.

```

+ handlers
+ contacts.cfc
+ model
+ Contact.cfc

```

ORM

Use the same settings as the previous example.

Contact.cfc

Change the Active Entity example by removing the inheritance.

```

/**
 * A cool Contact entity
 */
component persistent= "true" table= "contacts" {

    // Primary Key
    property name= "contactID" fieldtype= "id" column= "contactID" generator= "native" setter= "false" ;

    // Properties
    property name= "firstName" ormtype= "string" ;
    property name= "lastName" ormtype= "string" ;
    property name= "email" ormtype= "string" ;

    // validation
    this.constraints = {
        firstName = {required= true},
        lastName = {required= true},
        email = {required= true , type= "email" }
    };
}

```

Contacts Handler

That's right, go to the handler now, no need of data layers or services, we build them for you!

```

/**
 * I am a new handler
 */

```

```

component{
    // Inject a virtual service layer binded to the contact entity
    property name= "contactService" inject= "entityService:Contact" ;

    function index(event,rc,prc){
        prc.contacts = contactService.list(sortOrder= "lastName" ,asQuery= false);
        event.setView( "contacts/index" );
    }

    function editor(event,rc,prc){
        event.paramValue( "id" ,0);
        prc.contact = contactService.get( rc.id );
        event.setView( "contacts/editor" );
    }

    function delete(event,rc,prc){
        event.paramValue( "id" ,0);
        contactService.deleteByID( rc.id );
        getPlugin( "MessageBox" ).info( "Contact Removed!" );
        setNextEvent( "contacts" );
    }

    function save(event,rc,prc){
        event.paramValue( "id" ,0);
        var contact = populateModel( contactService.get( rc.id ) );
        var vResults = validateModel( contact );
        if( !vResults.hasErrors() ){
            contactService.save( contact );
            getPlugin( "MessageBox" ).info( "Contact Saved!" );
            setNextEvent( "contacts" );
        }
        else {
            getPlugin( "MessageBox" ).error(messageArray=vResults.getAllErrors());
            return editor(event,rc,prc);
        }
    }
}

```

Views

Here are the views as well, isn't it awesome that the views stay the same :), that means we did a good job abstracting our model and controllers.

index.cfm

```

<<cfoutput>
<h1> Contacts </h1>
#getPlugin( "MessageBox" ).renderit()#
#html.href(href= 'contacts.editor' ,text= "Create Contact" )#
<br><br>
<cfloop array= "#prc.contacts#" index= "contact" >
<div>
#contact.getLastName()# , #contact.getFirstName()# (#contact.getEmail()#) <br />
#html.href(href= 'contacts.editor.id.#contact.getContactID()#' ,text= "[ Edit ]" )#
#html.href(href= 'contacts.delete.id.#contact.getContactID()#' ,text= "[ Delete ]" ,onclick= "return confirm('Really Delete?')" )#
</div>
</cfloop>
</cfoutput>

```

editor.cfm

Check out our awesome `html` helper object. It can even build the entire forms according to the Active Entity object and bind the values for you!

```

<<cfoutput>
<h1> Contact Editor </h1>
#getPlugin( "MessageBox" ).renderit()#
#html.startForm(action= "contacts.save" )#
#html.entityFields(entity=prc.contact,fieldwrapper= "div" )#
#html.submitButton()# or #html.href(href= "contacts" ,text= "Cancel" )#
#html.endForm()#
</cfoutput>

```

Summary

The ColdBox [ORM:BaseORMService](#) and [ORM:VirtualEntityService](#) are so awesome as they give us a sense of using service layers virtually or concretely.