An Introduction to WireBox

www.coldbox.org

Covers up to version 1.7

Contents

What is WireBox	1
Installing WireBox	2
WireBox Injector	3
Configuring WireBox	3
Mapping your objects	3
Component Annotations	4
WireBox For More Than CFCs	5
Configuring your Components' Dependencies	5
Providers	6

WireBox is an enterprise ColdFusion (CFML) dependency injection (DI) and Aspect Oriented Programming (AOP) framework. WireBox's inspiration has been based on the concept of rapid workflows when building object oriented ColdFusion applications, programmatic configurations and simplicity.

With that motivation WireBox introduces dependency injection by annotations and conventions, which has been the core foundation of WireBox. If you don't know what that all means, don't worry! The important thing to know is that WireBox helps object-oriented CFML developers manage all the CFCs in their applications, so you can focus on the important stuff. Despite the power under the hood, WireBox has a strong focus on convention over configuration and its readable binder Domain Specific Language (DSL=fancy word for we know what to do with certain strings) makes for expressive and self-documenting object configuration that aims to simplify your life.

WIREBOX'S MAIN GOALS

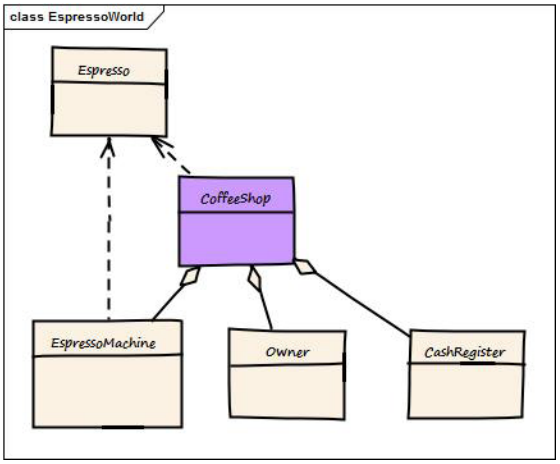
- Alleviate the need for custom object factories
- No more manual object creations and self-wiring
- Provide object life cycle and persistence
- Help your objects become more testable, extensible and mockable (That's a word!)
- Provide a centralized programmatic Binder for object mappings and configurations (no XML)
- Provide AOP capabilities
- Provide RAD workflows to object building and assembling
- Bring a nice smile to your face!

WireBox alleviates the need for custom object factories or manual object creation in your ColdFusion applications. It provides a standardized approach to object construction and assembling that will make your code easier to adapt to changes, easier to test, mock and extend. As software developers we are always challenged with maintenance and one ever occurring annoyance, change. Therefore, the more sustainable and maintainable our software, the more we can concentrate on real problems and make our lives more productive.

WireBox leverages an array of object, function, and property metadata annotations to make your object assembling, storage and creation easy as pie! We have leveraged the power of event driven architecture via object listeners or interceptors so you can extend not only WireBox but the way objects are analyzed, created, wired and much more. To the extent that our AOP capabilities are all driven by our AOP listener which decouples itself from WireBox code and makes it extremely flexible.

WireBox's most basic function is as an object factory, usually referred to as an Injector, that creates instances of not only your application's CFCs, but can also integrate and produce Java classes, web services, transient objects, constant values and even RSS feeds for you.

The next thing WireBox does for you is help manage the interdependencies of your objects.



Chances are, your objects aren't islands to themselves. They need references to other objects, services, DAOs, and even settings upon creation or even at runtime. With a bit of configuration, or better yet- annotation in the target CFC, WireBox will spin up all those dependencies for you as necessary and preload your object with everything it needs to be ready to go. All you need to do at run time is request the object.

The third basic feature WireBox provides is automatic persistence for any object you create.

You stop worry about trying to manually stuff objects into scopes after checking for their existence. Instead, tell WireBox how you want that object persisted and then simply ask your friendly DI framework for it whenever you need it and let it do the rest. Objects that are created new every time are called transients, which is the default behavior. The singleton pattern is the next most common persistence pattern. Objects such as services and DAOs are common singletons-- meaning there should only be one instance of those objects in the entire application and are usually stateless. WireBox doesn't stop there. It lets you persist objects in any ColdFusion scope or even any CacheBox provider.

Without a DI framework, you might have code like this scattered throughout your app:

```
// Create dependencies
var myUserService = new com.model.contacts.UserService();
var myUserDAO = new com.model.contacts.UserDAO(
                                        request.myDSN);
// Create user and init it with the service and DAO.
var myUser = new com.model.contacts.user(myUserService,
                                            myUserDAO);
// Set in additional settings required for the user object to
work
myUser.setUserDefaults(getSetting("userDefaults"));
// Persist user in session
session.loggedInUser = myUser;
// FINALLY ready to do stuff with it
session.loggedInUser.doCoolStuff();
```

This is how much simpler that code can be when you ask WireBox to create and persist your user object for you:

```
// Request user from WireBox, wired, persisted and ready to go
var myUser = injector.getInstance("user");
// Use it!
myUser.doCoolStuff();
```

Yeah, but how much configuration is required for THAT to work? None! It can all be done with a little metadata right in the user component. Check out what user.cfc might look like:

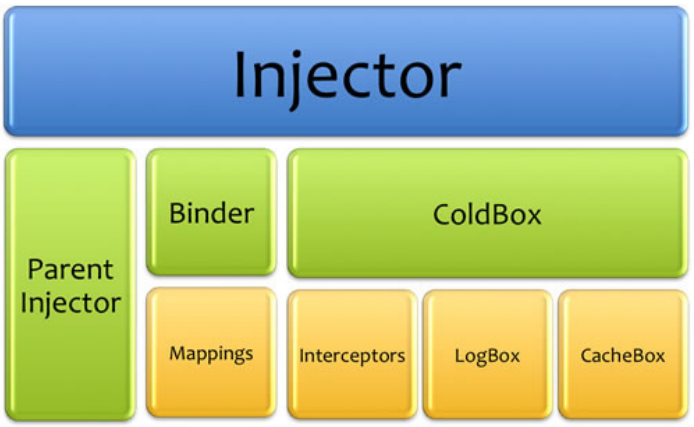
```
component name="user" scope="session" {
    property name="UserService" inject;
    property name="UserDAO" inject;
    property name="userDefaults"
             inject="coldbox:setting:userDefaults";
}
```

And what's great is there's no XML. Anywhere. Nada! Your objects can simply describe what they need in a natural and integrated way.

This is just scratching the surface of what WireBox can do for you. There's also amazing abstract ORM services loaded with handy utilities for working with Hibernate entities. Read on to find out how to get started.

INSTALLING WIREBOX

ColdBox-Bundled

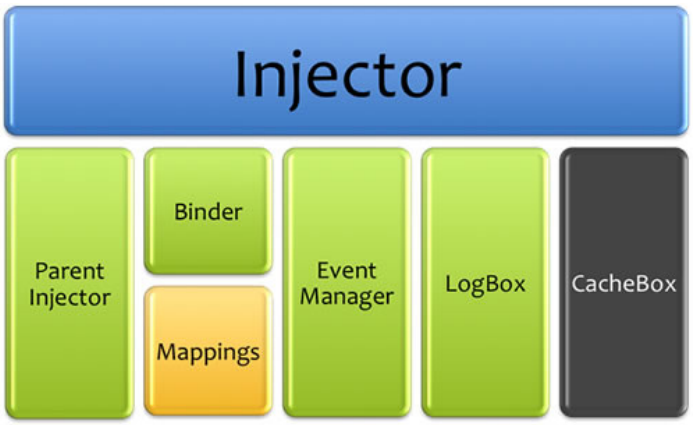


The most common way WireBox gets used is inside a ColdBox or ColdBox LITE application where it comes bundled standard issue. If you're using ColdBox, the WireBox engine (often referred to as the injector) is already created and deeply integrated with your app. Functions such as `getModel()` and `getWireBox()` are available directly inside your handlers, view, and layouts. The default configuration binder is a CFC called `WireBox.cfc` and located inside of the "`config`" folder.

`/config/WireBox.cfc`

To download the ColdBox MVC platform or ColdBox LITE (with bundled WireBox), visit `http://www.coldbox.org/download` and snag the latest version.

Standalone



You can also utilize the full WireBox goodness in standalone form very easily in two flavors:

- WireBox & AOP
- WireBox & AOP with CacheBox

Hit the same download link as above, and scroll down to "WIREBOX DI & AOP" for the latest version. Either place the "`wirebox`" folder in your webroot or create a CF mapping called "`wirebox`" that points to it.

When your application starts up, create an instance of `wirebox.system.ioc.Injector:`

```
wirebox = new wirebox.system.ioc.Injector();
```

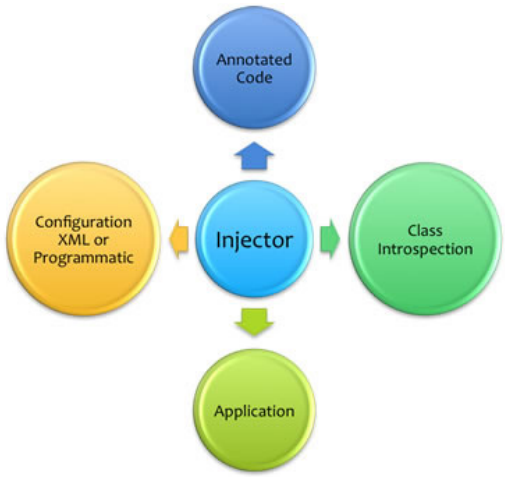
If you want to configure how WireBox runs, or specify explicit mappings, you can pass in the path to a config file on creation:

```
wirebox = new wirebox.system.ioc.Injector("path.to.config");
```

By default, WireBox has scope registration turned on which means it will set a reference to itself in application.wirebox automatically when created. That's how you can retrieve it later to ask it for objects.

Change any paths with `coldbox.system` to `wirebox.system` if you are using the standalone version. The majority of the examples will contain the wirebox.system path.

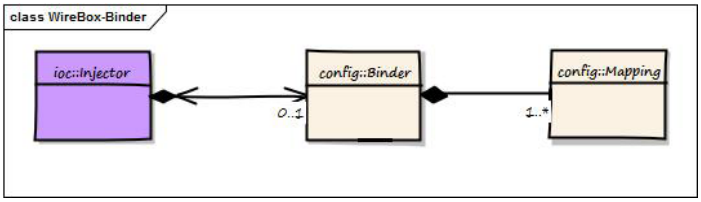
WIREBOX INJECTOR



WireBox bases itself on the idea of creating object injectors (`wirebox.system.ioc.Injector`) that in turn will produce and wire all your objects. You can create as many injector instances as you like in your applications, each with configurable differences or be linked hierarchically by setting each other as parent injectors. Each injector can be configured with a configuration binder or none at all like we saw before. ColdBox applications take advantage of deep integration and have no need to create the injectors as they are created for those applications.

More info: http://wiki.coldbox.org/wiki/WireBox.cfm#The_WireBox_Injector

CONFIGURING WIREBOX



Your configuration for WireBox is just a CFC file referred to as the "binder" since it is where you can bind actual objects to arbitrary IDs for your app to use. It also stores a collection of object mappings that represent your objects, metadata and their relationships within Wirebox.

If you are using WireBox inside of a ColdBox application, you can modify the default location of WireBox config file in your main ColdBox config with the following simple declaration:

```
wirebox = {
    binder = "path.to.config",
    singletonReload = true
};
```

The `singletonReload` setting instructs Coldbox to clear out all your singletons (like services and DAOs) on each request to help with development. Make sure this is turned off for production!

Your WireBox config file needs to extend "`coldbox.system.ioc.config.Binder`" and only needs one method: `configure()`. Here's a sample WireBox binder config with just the basics:

```
component extends="coldbox.system.ioc.config.Binder" {
    function configure() {
        wireBox = {
            scanLocations = ["model","myCFCs"]
        };
        map("ProductService").to("model.products.
                                  ProductService");
        map("productDAO").to("model.products.ProductDAO");
        mapDirectory("coolServices");
    }
}
```

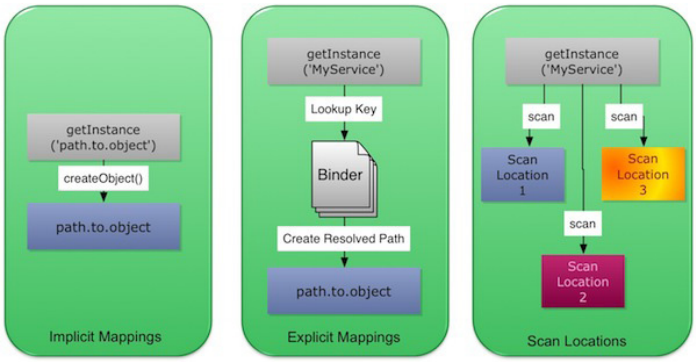
More info: http://wiki.coldbox.org/wiki/WireBox.cfm#Configuring_WireBox

MAPPING YOUR OBJECTS

Let's talk about what that code does. One of WireBox's core functions is an object factory. Basically that's an object which you delegate to in order to create instances of the other objects your application needs. WireBox keeps track of all the objects it can create using binder mappings. You can have as many mappings as you want in Wirebox and each one keeps track of how to create that object, how to persist it, and the object's dependencies.

Asking WireBox to create objects for you does is so much more than a simple replacement for `createObject()` or new. It allows WireBox to handle the persistence of that object as well as injecting all the required dependencies, framework plugins, settings, etc that the object needs to be fully functioning. It shifts the responsibility from you to itself so you just ask for what you need and WireBox takes care of the rest! That's why WireBox is also known as an IoC (Inversion of Control) container, because you are relinquishing control of object creation to WireBox instead of you doing it all by your lonesome.

You can tell WireBox what mappings you'd like it to be aware of, or it can create them as it goes. There's several ways to accomplish the same thing in WireBox, and they're all just as valid We'll cover them below and you can choose the method that works best for your style and your application.



Implicit Mappings

The easiest way to request an instance of an object from WireBox without the need of any configurations is to simply reference it with the full instantiation path just like you would with `createObject()` or the `new` operator. So if you typically had code like this:

```
myModel = new model.shoppingCart.Discount();
```

You could simply replace with this code:

```
// standalone
myModel = injector.getInstance("model.shoppingCart.
Discount");
// coldbox
myModel = getModel("model.shoppingCart.Discount");
```

The `getModel()` method is automatically available inside of handlers, and views if you're in a ColdBox application, which delegates to `wirebox.getInstance()`. WireBox standalone could look something like this:

Explicit Mappings

The next way to tell WireBox about your CFCs is to create explicit mappings in the config file. The configuration Binder has a very handy DSL of chainable methods you can use to expressly declare mappings, how you want them created, and a simple ID to refer to them in your app.

```
map("myCoolService").to("com.my.services.CoolService");
```

If you want the mapping ID to simply be the name of the component you can just shorten that to this:

```
mapPath("model.services.coolService");
```

That's not all. Look at some these cool options you can chain together:

```
mapPath("model.services.CoolService").asSingleton();
mapPath("model.security.user").into(this.SCOPES.SESSION);
map("news")
    .to("model.pods.news")
    .inCacheBox(timeout=20, provider="Couchbase");

// standalone
myModel = injector.getInstance("CoolService");
// coldbox
myModel = getModel("CoolService");
```

If you have a whole directory of CFCs that you want mapped, this line will recursively search the directory at startup and map every CFC it finds using the component name as the mapping ID.

```
mapDirectory( "model.products" );
```

Scan locations

The final way for WireBox to find your components is by using scan locations. Use this if you don't want to specify the full component path in your code and you don't want to have to map every component in the config. The config lets you set an array of scan locations, or base directories that should be searched. Just like `/views` and `/handlers` are the default convention folder that views and handlers are searched in, think of scan locations as a models convention. If you're using WireBox inside of ColdBox, the model directory will automatically added as default scan location.

```
wirebox.scanLocations = ["model"];
```

That means that give the following file:

```
\model\company.cfc
```

You can create it like so:

```
injector.getInstance("company");
getModel("company");
```

For files in a subdirectory, specify the partial path after the root of the scan location.

```
\model\security\rule.cfc
injector.getInstance("security.rule");
getModel("security.rule");
```

COMPONENT ANNOTATIONS

You may have noticed that some of those methods like explicit mappings and scan locations don't give you way to specify things like persistence in the config. Don't worry-- WireBox solves this by inspecting the component's metadata at creation for special annotations you can place there. Here's a non-inclusive list of examples:

```
// Only create a single instance of this component
component singleton {}
// Create one of these per session and store in the
session scope
component scope="session" {}
// Only create this component once every 10 minutes and store
it in the default cache
component cache=true cacheTimeout=10 {}
```

WIREBOX FOR MORE THAN CFCS

So far all I've shows are examples that create CFML components, but WireBox is actually capable of much more. It can be used to help your application create Java classes, RSS feeds, Web Service objects, constants and custom DSLs. Just use the same binder methods as our explicit mappings use:

```
map("buffer").toJava("java.lang.StringBuffer");

// cache some google news and refresh it every 20 minutes
map("latestNews")
    .inCacheBox(timeout=40, provider="Couchbase");
    .toRSS("http://news.google.com/news?output=rss")

map("myWS").toWebservice("http://myapp.com/app.cfc?wsdl");

map("DBPassword").toValue("$up3r$3cr3t");

map("Logger").toDSL("logbox:root");
```

CONFIGURING YOUR COMPONENTS' DEPENDENCIES

So we've talked about how to make WireBox aware of your objects so it can create them for you, but how do we set up the interdependence between objects? For instance, you may have a Person object who needs access to a PersonService and PersonDAO. As usual, there's several ways to accomplish this and you can choose the way that works the best for you.

Injection Types

MIXIN (AUTOWIRE)

Mixin injection, also known as autowiring, is probably the easiest and least intrusive way to add dependencies to an object. It uses CFML's dynamic nature to inject reference variables into your target component without you creating any special methods or arguments to receive them. All you need to do is set up some properties in your target component that tells Wirebox what it needs. These properties minimally need to have an attribute called "inject" whose value can be a mapping ID, or one of several special annotation namespaces. Let's assume our **PersonService** and **PersonDAO** already have mappings configured, meaning WireBox already knows how to create them. The Person object only needs these two extra lines:

```
component name="Person" {
    property name="personService" inject="PersonService";
    property name="personDAO" inject="id:PersonDAO";
}
```

By default, this will place references in the variables scope named after the "name" attribute. To customise the target scope, simply use the "scope" attribute like so:

```
property name="personService" inject="personService"
         scope="this";
property name="personDAO" inject="personDAO"
         scope="instance";
```

Property injection is a great self-documenting way to specify dependencies right there in your objects. The only catch to watch out for is they aren't injected until AFTER your `init()` method is called. Thus, it can be used safely when dealing with object circular references. If you need to use them to get your object ready for use, create an `onDIComplete()` method. It will automatically be called after injection by convention and you can use the injected dependencies in the method:

```
function onDIComplete() {
    cachedRoles = personService.getAllRoles();
}
```

CONSTRUCTOR

The next method of getting dependencies into your objects is by having WireBox pass them into your constructor as method arguments. The default constructor is `init()` and all you need to do is add an "inject" attribute that follows the same rules as the property annotations above.

```
component name="personBean" {
    function init(
            required any personService inject="personService",
            required any personDAO inject="personDAO"
            ) {
        variables.personService = arguments.personService;
        variables.personDAO = arguments.personDAO;
        return this;
    }
}
```

Note the "`required`" keyword and return type are necessary in cfscript to prevent parser errors. You can also achieve constructor inject in cfscript with method annotations:

```
component name="personBean" {
    /**
    * init
    * @personService.inject personService
    * @personDAO.inject personDAO
    */
    function init(personService, personDAO) {
        variables.personService = arguments.personService;
        variables.personDAO = arguments.personDAO;
    }
}
```

One benefit of constructor injection is that you get to use your dependencies right there in your `init()` method, but you do need to set them into a persistent scope inside your component (this, variables, etc) if you plan on keeping them around for subsequent method calls. Also note, that circular dependencies are only possible via constructor injection if you use a provider (covered later)

SETTER

If you like to have setter methods for all your properties, you can create an annotated setter for each dependency and WireBox will call it automatically to set in the dependencies upon creation of your component. Sticking with our Person object example, here's what that would look like:

```
component name="Person" {
    function setPersonService(personService) inject {
        variables.personService = arguments.personService;
    }
    function setPersonDAO(personDAO) inject {
        variables.personDAO = arguments.personDAO;
    }
}
```

Setter methods must start with the text "set". WireBox uses the remainder of the method name to determine the mapping ID to inject. You can also specify the mapping ID as the value of the "inject" attribute but the method name will still control the name of the incoming argument.

```
function setCoolStuff(CoolStuff) inject="personService" {
    variables.personService = arguments.CoolStuff;
}
```

Setters are called after `init()`, so you'll need to wait until `onDIComplete()` to access the injected properties.

Injection DSL

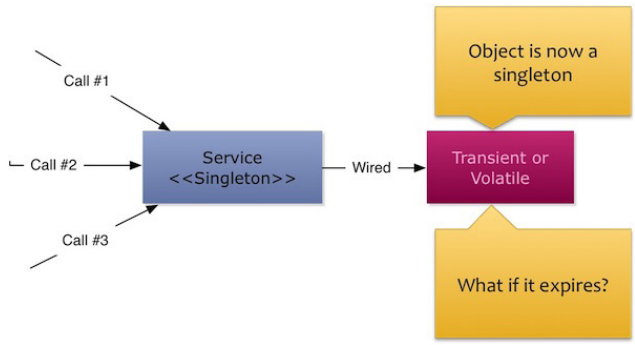
All our examples so far have been injecting mappings using their ID. What if you want to inject a reference to ColdBox itself, LogBox loggers, or Cachebox caches from the framework? There is an incredibly powerful set of annotation namespaces can give you access to just about anything you could want from the framework. These can be used in any of the injection styles above as the value of your "inject" property. Consider these examples:

```
// Get a reference to the ColdBox controller
inject="coldbox"
// Get a reference to the WireBox injector
inject="wirebox"
// Inject the "myCache" CacheBox provider
inject="cachebox:myCache"
// Specific logger for this component (Don't replace "{this}",
leave it just like that
inject="logbox:logger:{this}"
// How about a logger named "Fred"?
inject="logbox:logger:Fred"
// This is a setting defined in the main ColdBox config
inject="coldbox:setting:mySetting"
// Get the MessageBox plugin
inject="coldbox:plugin:MessageBox"
// An ORM virtual entity service
inject="entityService:User"
```

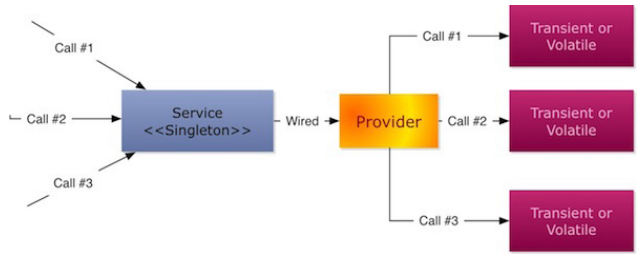
How cool is that? We won't cover it here, but WireBox also lets you register your own custom namespaces and define a builder CFC to create whatever you want.

More info: http://wiki.coldbox.org/wiki/WireBox.cfm#Injection_DSL

PROVIDERS



Sometimes there is a reason not to inject an object directly into another object. Setting `loggedInUser` directly into the variables scope of myService means that myService has a single "hard reference" to `loggedInUser`. If myService is a singleton and `loggedInUser` is a session-scope object this would be known as scope-widening injection since singletons live longer than a session. This hard reference would also prevent myService from getting the correct `loggedInUser` from the current session which is bad. Another scenario is if you were to inject objects that weren't ready to be used yet because the framework is still booting up or you just want to delay the construction of objects or even to use circular dependencies via constructor arguments. These scenarios can be remedied but using providers.



Providers are a loosely-coupling object that defers resolution of the dependency until you need it. When you inject a provider, you don't get the actual object, but a placeholder object that is capable of retrieving the correct version of the real object when you need it.

Let's look at a simple example of using a provider to inject a personBean into a component.

That's it! All you need is to use the provider namespace prior to the mapping ID. The provider comes with a single `get()` method to produce the actual dependency:

```
component {
    property name="personBeanProvider"
             inject="provider:personBean";
}
```

Providers also employ `onMissingMethod()` to directly proxy method calls through to the actual object being provided:

```
personBeanProvider.get().sayHello();
```

Every time you use a provider, it goes back to WireBox and asks for the object again so you never have to store a direct reference to the dependency, just the empty provider shell.

```
personBeanProvider.sayHello();
```

WireBox is completely documented in our online wiki located at
http://wiki.coldbox.org/wiki/WireBox.cfm

For API docs to see class definitions and method signatures, please visit the API docs located at http://www.coldbox.org/api

We have an active Google Group with hundreds of subscriber located at

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

Our official code repository is on GitHub. Please favorite us and feel free to fork and submit pull requests.

https://github.com/ColdBox/coldbox-platform

WIREBOX
By ColdBox

An Introduction to WireBox

Ortus Solutions
A Software Revolution

CacheBox is professional open source backed by Ortus Solutions, who provides training, support & mentoring, and custom development.

Support Program

The Ortus Support Program offers you a variety of Support Plans so that you can choose the one that best suit your needs. Subscribe to your plan now and join the Ortus Family!

With all of our plans you will profit not only from discounted rates but also receive an entire suite of support and development services. We can assist you with sanity checks, code analysis, architectural reviews, mentoring, professional support for all of our Ortus products, custom development and much more!

For more information visit
www.ortussolutions.com/services/support

Our Plans	M1	Entry	Standard	Premium	Enterprise
Price	**$199**	**$2099**	**$5699**	**$14099**	**$24599**
Support Hours	2 4 tickets	10 20 tickets	30 60 tickets	80 160 tickets	150 300 tickets
Discounted Rate	**$185/hr**	**$180/hr**	**$175/hr**	**$170/hr**	**$160/hr**
Renewal Price	**$199/month**	**$1800/year**	**$5250/year**	**$13600/year**	**$2400/year**
Phone/Online Appointments	✔	✔	✔	✔	✔
Web Ticketing System	✔	✔	✔	✔	✔
Architectural Reviews	✔	✔	✔	✔	✔
Hour Rollover		✔	✔	✔	✔
Custom Development		✔	✔	✔	✔
Custom Builds & Patches			✔	✔	✔
Priority Training Registration			✔	✔	✔
Development & Ticket Priority			✔	✔	✔
Response Times	1-5 B.D.	1-3 B.D.	1-2 B.D.	< 24 hr	< 12 hr
Books, Trainings, Product Discounts	0%	5%	10%	15%	20%
Free Books	0	0	1	3	5

ProfileBox

COLDBOX METRICS & PROFILING

Leveraging the power of Integral's FusionReactor, fine tune, optimize and debug your ColdBox applications with absolute ease!

Find out more at www.ortussolutions.com/products/profilebox

Ortus Solutions
A Software Revolution

fusion reactor™

DESIGNED FOR

ColdBox ContentBox

www.coldbox.org | www.ortussolutions.com

First Edition - October 2013