An Introduction to CacheBox

www.coldbox.org

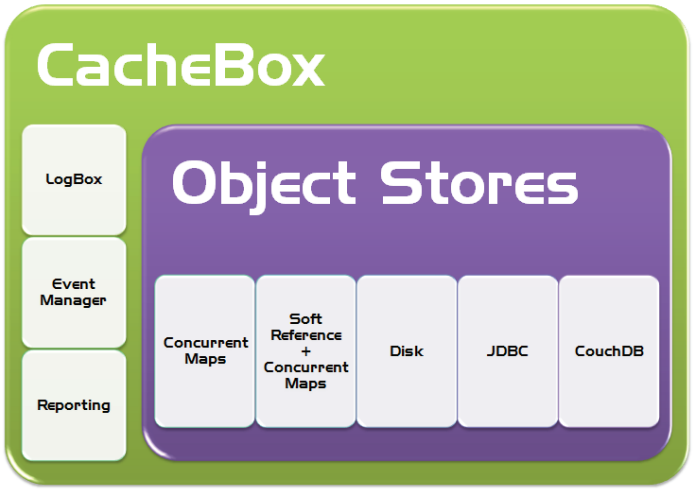
Covers up to version 3.7.0

Contents

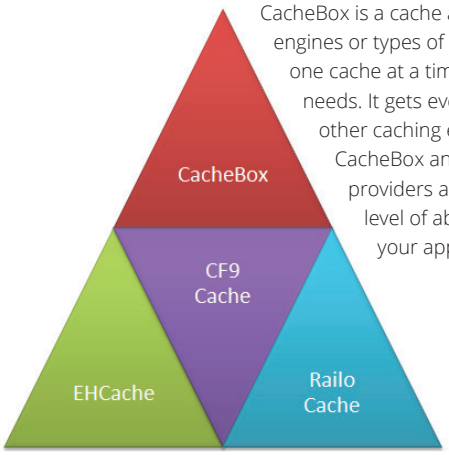
CacheBox is an enterprise cache engine, aggregator and API for ColdFusion applications. It can be used on its own as a standalone library and also comes bundled with the ColdBox MVC Platform.

ENTERPRISE CACHING ENGINE

CacheBox behaves as an in-memory cache designed for handlers, plugins, events, views fragments and any other objects or data you so desire. CacheBox has a small memory footprint, and a multithreaded core to be light and fast. It also features a click web console that allows you to report on your cache's hit/miss and memory performance as well as view the contents and run commands against any registered cache. There is even an event model you can tie into for cache element and lifecycle events and integrated logging via LogBox. Regular timeouts are supported as well as last accessed timeouts which lets you keep oft-requested keys in favor of stagnant ones. You can fully configure the maximum number of items, JVM memory thresholds, as well as custom eviction policies like LRU, LFU, and FIFO. The most common cache is a Java concurrent hashmap soft reference store. This avoids memory contention and protects from out of memory errors, but there are a number of other stores you can choose from such as the disk store or even a JDBC store.



CACHE AGGREGATOR & API



CacheBox is a cache aggregator, in which you can aggregate different caching engines or types of the same engine into one single umbrella. You don't just get one cache at a time! This let's you easily segregate your application's caching needs. It gets even better though- CacheBox has a provider API that allows other caching engines like EHCache or Memcached to be aggregated by CacheBox and presented alongside your other caches. These cache providers also share the same interaction API and thus gives you a nice level of abstraction, higher extensibility and flexibility when planning your applications or trying to scale them out.

INSTALLATION

ColdBox-Bundled

If you are using the ColdBox Platform, you don't need to do anything--CacheBox is included for you and the framework creates the cache factory for you. The default config is a CFC called `CacheBox.cfc` and located inside of the "config" folder.

`/config/CacheBox.cfc`

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

Standalone

If you would like to use CacheBox on its own, visit the download page `http://www.coldbox.org/download` and download the standalone CacheBox library. After unzipping the library, the easiest way to install it is to place the "`cachebox`" folder in your webroot. A more secure method is to leave that folder outside the webroot and create a "`/cachebox`" ColdFusion mapping that points to it.

When your application starts up, create an instance of `cachebox.system.cache.CacheFactory:`

```
cachebox = new cachebox.system.cache.CacheFactory();
cache = cachebox.getDefaultCache();
```

If you want to configure how CacheBox runs, or create named caches, you can pass in the path to a config file on creation:

```
// Create the config file
config = new cachebox.system.cache.config.
CacheBoxConfig(CFCConfigPath="path.to.config");
// Pass our config file into CacheBox
cachebox = new cachebox.system.cache.CacheFactory(config);
```

By default, CacheBox has scope registration turned on which means it will set a reference to itself in `application.cachebox` automatically when created. That's how you can retrieve it later to get your caches.

Change any paths with `coldbox.system` to `cachebox.system` if you are using the standalone version. All the examples will use the standalone configuration.

CONFIGURATION

CacheBox will run with no config file by loading a single default cache based on the concurrent soft reference store with a max size of 300 items, which we name lovingly '**default**'. Customize your install or add additional caches by modifying your `CacheBox.cfc` config file. The config are just a simple CFC with a single `configure()` method that creates a struct called "**cachebox**" that contains the settings.

When used inside of ColdBox, the default config file is this:

```
function configure() {
    cacheBox = {
        defaultCache = {
            objectDefaultTimeout = 120,
            objectDefaultLastAccessTimeout = 30,
            useLastAccessTimeouts = true,
            reapFrequency = 2,
            freeMemoryPercentageThreshold = 0,
            evictionPolicy = "LRU",
            evictCount = 1,
            maxObjects = 300,
            objectStore = "ConcurrentStore",
            coldboxEnabled = true
        },
        caches = {
            template = {
                provider = "coldbox.system.cache.providers.
                            CacheBoxColdBoxProvider",
                properties = {
                    objectDefaultTimeout = 120,
                    objectDefaultLastAccessTimeout = 30,
                    useLastAccessTimeouts = true,
                    reapFrequency = 2,
                    freeMemoryPercentageThreshold = 0,
                    evictionPolicy = "LRU",
                    evictCount = 2,
                    maxObjects = 300,
                    objectStore = "ConcurrentSoftReferenceStore"
                }
            },
            listeners = []
        }
    };
}
```

The defaultCache struct configures the cache named "**default**". This is a required cache and it must use the CacheBox provider. This cache is used to store pieces of the framework like handlers and interceptors if used by ColdBox or for any type of data if using standalone.

Additional named caches are declared in the "**caches**" struct. The template cache is used by the ColdBox framework for view and event caching which we'll cover next. If you're using ColdBox's built-in caching, you should monitor the size and performance of the cache and tune the maxObjects and default timeout as necessary.

Additional caches can specify what provider they will use and can look like this:

```
myCache = {
    // Use CF's EHCache implementation
    provider = "coldbox.system.cache.providers.CFProvider"
}
```

Note: Different cache providers can have different properties. Check the docs for the provider you're using.

More Info: `http://wiki.coldbox.org/wiki/CacheBox.cfm#Cache_Providers`

COLDBOX VIEW CACHING

The ColdBox Platform is tightly integrated into CacheBox. It has some built-in features for caching bits of your application without even having to touch the cache engine. One of those features is view fragment caching. When declaring the display view in your handler, simply add `cache="true"` in like so:

```
event.setView(view="general/index", cache=true);
```

The rendered output from that view will be placed in the "**template**" cache and won't be rendered again as long as the item remains in the cache. I control how long a view should be cached, use the `cacheTimeout` parameter. If there variations in the view-- like different languages on your site, you can supply a `cacheSuffix` to ensure each unique version of the view gets cached separately.

```
event.setView(view="products/widget",
              cache=true,
              cacheTimeout=60,
              cacheSuffix=getfwLocale());
```

Remember, cached views are shared by the entire application so don't cache a view that has completely unique information on it like the logged in user's name or items in a shopping cart.

COLDBOX EVENT CACHING

The next ColdBox Platform integration we'll cover is event caching. This is as simple as view fragment caching, but broader in scope. This time we're caching the entire output of an event, including any sub events it ran, any and all views that may have been rendered, as well as the status code, content type, and encoding. To enable event caching, turn on the similarly-named setting in the ColdBox configuration CFC.

```
coldbox = {
    eventCaching = true
};
```

Now just add `cache="true"` to the methods in your handler that you would like to be cached.

```
function productDetail(event,rc,prc) cache=true {
    prc.productInfo = productService.getInfo(rc.id);
    event.setView("products/productDetail");
}
```

ColdBox makes a hash of the request collection at the beginning of the request and this is used to create a unique cache key. That means that a separate item will be created in the cache for each unique combination of URL/Form parameters. That means that each of the following URLs would be cached separately:

```
mysite.com/products/productDetail?id=1
mysite.com/products/productDetail?id=2
mysite.com/products/productDetail?id=2&language=french
```

Remember to consider how many unique ways your event might be hit and tune your template cache size accordingly. Both view and event caching support a `cacheLastAccessedTimeout` parameter designed to help the cache clear out items which are not being hit, while retaining other items that are used a lot.

```
// Cache this event for 45 minutes as long as it's hit at
least once every 5 minutes
function productDetail(event,rc,prc) cache=true
         cacheTimeout=45 cacheLastAccessedTimeout=5 {
    prc.productInfo = productService.getInfo(rc.id);
    event.setView("products/productDetail");
}
```

CACHE API

Every cache in CacheBox implements a standard API for getting, setting, looking up, and clearing items. You can use this interface to to manually store whatever your application needs from queries, lookup data, or results of slow web services.

If you're using CacheBox standalone and want to get a cache, get the cache factory out of its registered scope and ask it for the cache by name.

```
myCache = application.cachebox.getCache("myCache");
```

If you're inside a ColdBox handler or view, you can use the `getColdboxOCM()` method to retrieve your cache. Not passing in a cache name will give you the default cache.

```
myCache = getColdboxOCM("myCache");
```

Another way to do it that also works in model objects is to ask WireBox to inject a reference to your cache into a component by placing an injection property at the top of the CFC.

```
component {
    property name="myCache" inject="cachebox:myCache";
}
```

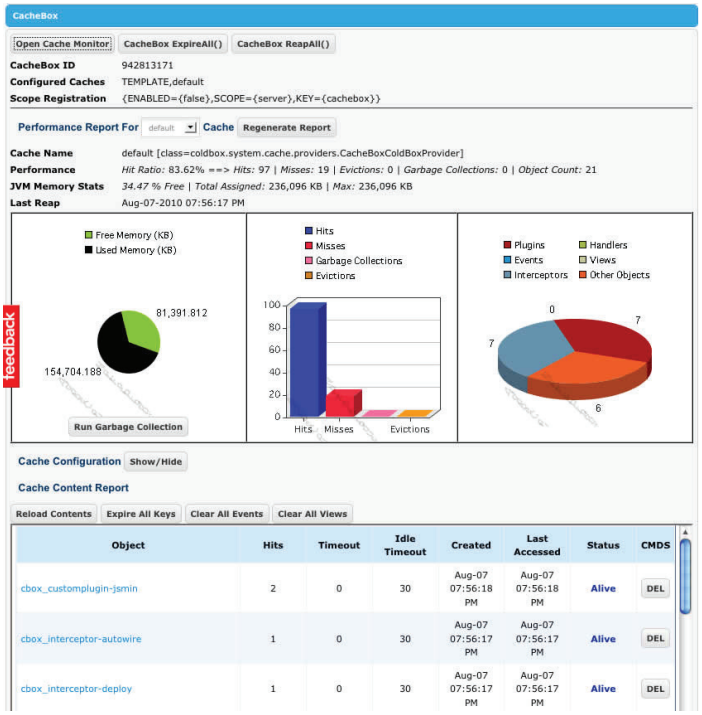
Regardless of how you get a reference to the cache, using it is the same. Here are some common methods you'll want to be familiar with.

```
myCache.set("key","value");
myCache.lookup("key");
myCache.get("key");
myCache.getKeys();
myCache.getStats();
myCache.clear("key");
myCache.clearAll(async=true);
```

REPORTING

There is also a suite of reporting tools to help with development and cache tuning. CacheBox keeps statistics for hits and misses as well as last-accessed timestamps and timeouts. Taking a peek inside your caches is pretty easy too. If you're using CacheBox inside the ColdBox framework, enable debug mode for the server by turning on the debugMode setting in your config, or enable it just for yourself by adding `?debugMode=1` to your URL and expand the "**CacheBox Monitor**" panel. If using standalone, you will put a call to our reporting custom tag wherever you want the cache report output.

```
<cfimport prefix="cachebox"
        taglib="/cachebox/system/cache/report">
<cachebox:monitor cacheFactory="#application.cacheBox#"/>
```



Change the drop down at the top of the report to choose which cache you want to view data for. High-level stats are available for the cache at the top of the panel like total hits, misses, and evictions. The graphs show you JVM memory usage as well as your hit and miss ratios. There is a "**Show/Hide**" button to show you the configuration for that cache.

At the bottom is a list of all the keys in the cache. The list includes key-level stats, timeout, created and last accessed times. Click the key name, to open up the contents of that item in a popup. There are a number of commands you can run on the cache like removing/expiring keys, clearing cached events or views, and reaping the cache.

Make sure you check this panel while performing load tests to keep an eye on your cache performance. More hits than misses, or large number of eviction can be a sign that your cache is too small or timeouts are too long.

CACHE PROVIDERS

We talked about cache providers earlier. Each cache uses a CFC called a provider to proxy to the used caching engine that actually persists its items. Here's a rundown of the providers that ship with CacheBox.

CacheBox provider - This is a custom cache engine written in hybrid Java/CFML that can use several different object stores. Please see the next section for the object stores.

ColdFusion's cache - EHCache

Railo's cache - configured in the Railo admin-- EHCache, RamCache, Memcached, etc.

Couchbase Server - distributed NoSQL document store and caching system (This provider is available on ForgeBox)

Mock Provider - Simple bare-bones implementation that keeps no stats. Use this for testing.

Write your own! - That right, you can extend CacheBox to connect to whatever underlying caching engine you want by writing a CFC that implements the ICacheProvider interface.

CACHEBOX PROVIDER STORES

When using the CacheBox provider, you can also choose the object store you want to use. Here's a list of the stores available to the CacheBox provider.

Concurrent Store - uses Java concurrent hash map to allow quick multi-threaded access to items.

Concurrent Soft Reference Store - Same as above, but with Java soft reference objects which will allow the Java garbage collector to recover memory if it needs.

Disk Store - Stores items as files on disk. Can be used for a distributed store if multiple servers point to the same network folder.

JDBC Store - Stores items as records in a DB table. Can also be used for a distributed store.

Blackhole Store - Doesn't really store anything-- just for testing how your app will work without any caching.

Write your own! - You can extend the CacheBox provider by writing a CFC that implements the IObjectStore interface.

CACHEBOX EVENT MODEL

CacheBox also sports a very nice event model that can announce several cache life cycle events and factory life cycle events. You can listen to these events and interact with caches at runtime very easily, whether you are in standalone mode or within a ColdBox application. Each event execution also comes with a structure of name-value pairs called **interceptData** that can contain objects, variables and all kinds of data that can be useful for listeners to use. This data is sent by the event caller and each event caller decides what this data sent is. Please note that each cache provider implementation announces different events as well.

Here are some common CacheBox Factory events:

- `afterCacheRegistration`
- `afterCacheRemoval`
- `beforeCacheReplacement`
- `afterCacheFactoryConfiguration`
- `beforeCacheFactoryShutdown`
- `beforeCacheShutdown`
- `afterCacheShutdown`

Here are some common CacheBox Provider events:

- `afterCacheElementInsert`
- `afterCacheElementRemoved`
- `afterCacheElementUpdated`
- `afterCacheClearAll`
- `afterCacheElementExpired`

Here is a listener example:

```
component {
    function configure(cacheBox,properties) {
        variables.cacheBox = arguments.cacheBox;
        variables.properties = arguments.properties;
        log = variables.cacheBox.getLogBox().getLogger(this);
    }
    function afterCacheElementRemoved(event, interceptData){
        var cache = arguments.interceptData.cache;
        var key = arguments.interceptData.cacheObjectKey;
        log.info("The cache #cache.getName()# just removed the
                  key -> #key#");
    }
}
```

More Info: http://wiki.coldbox.org/wiki/CacheBox.cfm#CacheBox_Event_Model

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

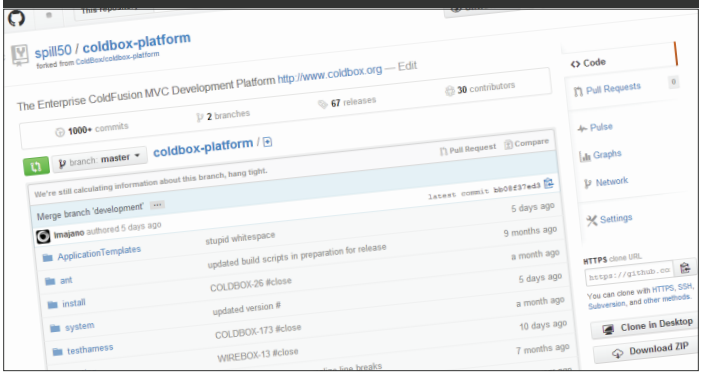
For API docs to see class definitions and method signatures, please visit the API docs located at http://www.coldbox.org/api

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



An Introduction to CacheBox

Ortus Solutions
A Software Revolution

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

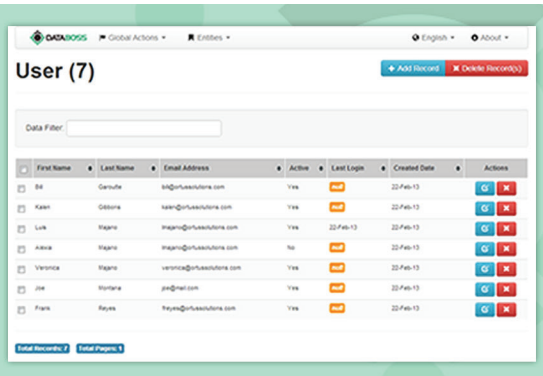
Support Program

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



"Build dynamic ORM administrators in seconds"

DATABOSS
Dynamic Administrator

www.data-boss.com

www.coldbox.org | www.ortussolutions.com

First Edition - October 2013