

[← Back to Dashboard](#)

CacheBox: The Enterprise ColdFusion Caching Engine, Aggregator and API

Contents

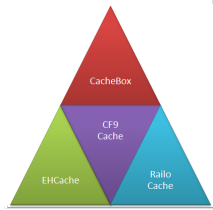
Covers up to version 1.2.0

Introduction

CacheBox is an enterprise caching engine, aggregator and API for ColdFusion applications. It is part of the ColdBox 3.0.0 Platform but it can also function on its own as a standalone framework. Phew! That's a mouthful, so let's talk about each one of the concerns that CacheBox tackles.

CacheBox: The 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. It gives you built in logging (via [LogBox](#)), an event model, synchronization, shutdown/startup procedures, reporting, interaction consoles and best of all a cache agnostic API. You can then build your applications based on an abstract API and then be able to configure the caches for your applications at runtime. This gives you greater flexibility and scalability when planning and writing your applications as they can be targeted for ANY CFML engine or version.

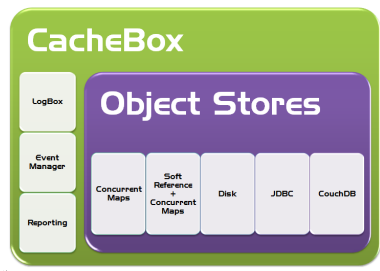


CacheBox gives you the ability to manage and create your own, what we call *Cache Providers* that talk to different caching engines that you would like to configure in your applications. By aggregating them within CacheBox you get the added benefit of a rich event model for all caches to share, reporting and debugging. We have a great cache debugger where you can visualize all your configured caches right from within your application. You can see the performance of the cache, the contents and even issue commands to the targeted cache provider.

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.

CacheBox : The Enterprise Caching Engine

Caching has been a central concern in the ColdBox Platform since its very first version. We have always been concerned with mission critical applications, scalability and the ability to provide granular caching aspects to ColdBox applications. With CacheBox we decided it was time to open up the library for usage to all ColdFusion applications, not even if they are built on ColdBox (Shame on you!).



Foremost, CacheBox behaves as an in-memory cache designed for handlers, plugins, events, views fragments and any other objects or data you so desire. It has various tuning parameters such as default object timeout, default last access timeout, maximum objects to have in cache, a JVM memory threshold, a reaping frequency, eviction policies, an event model and so much more.

We also make use of object stores (inspired by EHCACHE) for the CacheBox caching engine. This means that CacheBox can be configured to store its cached objects in different locations other than memory. This gives us great flexibility because we offer a wide gamut of storage options plus the concept of actually building your own storages. Currently we offer object stores based on concurrency classes, java soft references (memory sensitive), file storage, JDBC replication, and more coming soon.

- [CacheBox - The Enterprise ColdFusion Caching Engine, Aggregator and API](#)
 - [Introduction](#)
 - [CacheBox - The Cache Aggregator & API](#)
 - [CacheBox - The Enterprise Caching Engine](#)
 - [CacheBox Features](#)
 - [Downgrading CacheBox](#)
 - [Installing CacheBox](#)
 - [System Requirements](#)
 - [Internal Resources](#)
 - [Caching Concepts](#)
 - [Caching Considerations](#)
 - [Cache Eviction](#)
 - [Definitions](#)
 - [Event Manager Reference](#)
 - [Cache Providers](#)
 - [Single Instance/In-Process](#)
 - [Single Instance/Out-Process](#)
 - [Replicated](#)
 - [Distributed](#)
 - [CacheBox Architecture](#)
 - [Cache Factory](#)
 - [Cache Config](#)
 - [Cache Manager](#)
 - [ColdBox](#)
 - [LogBox](#)
 - [TemplateProvider](#)
 - [TemplateStore](#)
 - [TemplateStore](#)
 - [TemplateStore](#)
 - [AbstractEvictionPolicy](#)
 - [ColdBox ApplicationCache](#)
 - [Creating CacheBox](#)
 - [Common Cache Factory Methods](#)
 - [Cache Cleanup/Reaping](#)
 - [CacheBox Configuration](#)
 - [CacheBox DSL](#)
 - [LogBox Config](#)
 - [Scope Registration](#)
 - [Default Cache](#)
 - [CacheBox](#)
 - [Listeners](#)
 - [CacheBox Config Object](#)
 - [XML Configuration](#)
 - [ColdBox Application Configuration](#)
 - [Prerequisites Configuration](#)
 - [XML](#)
 - [Cache Providers](#)
 - [CF Providers](#)
 - [Properties](#)
 - [Rollin Provider](#)
 - [Properties](#)
 - [Mock Provider](#)
 - [Properties](#)
 - [CacheBox Provider](#)
 - [Properties](#)
 - [CacheBox Object Stores](#)
 - [ConcurrentStore](#)
 - [Properties](#)
 - [DiskStore](#)
 - [Properties](#)
 - [JDBCStore](#)
 - [Properties](#)
 - [RiakStore](#)
 - [Properties](#)
 - [CacheBox Eviction Policies](#)
 - [Using Your Own Policy](#)
 - [CacheBox Event Model](#)
 - [CacheBox Events](#)
 - [CacheBox Provider Events](#)
 - [CacheBox Provider CacheBoxColdBoxProvider Events](#)
 - [CacheBox Provider Events](#)
 - [Cache Listeners](#)
 - [Cache Reporting](#)
 - [CacheBox Report Store](#)
 - [Skin Attributes](#)
 - [Tag Callers](#)
 - [Skin Templates](#)
 - [Report Handler](#)
 - [Action Commands](#)
 - [ColdBox Application Enhancements](#)
 - [ColdBox Caches](#)
 - [Resources](#)

Cache Configuration [show/hide](#)

Cache Content Report

Object	Hits	Timeout	Idle Timeout	Created	Last Accessed	Status	CMDs
cbx_customplugin-gsm	2	0	30	Aug-07 07:56:18 PM	Aug-07 07:56:18 PM	Alive	DEL
cbx_interceptor-outside	1	0	30	Aug-07 07:56:17 PM	Aug-07 07:56:17 PM	Alive	DEL
cbx_interceptor-deploy	1	0	30	Aug-07 07:56:17 PM	Aug-07 07:56:17 PM	Alive	DEL

CacheBox Features

- **Cache Aggregator**
 - Ability to aggregate different caching engines
 - Ability to aggregate different configurations of the same caches
 - Rich aggregation event model
 - Granular logging
- **Fast and Simple to use**
 - Fast ColdFusion and Java hybrid cache
 - Simple API and configuration parameters
 - Small Footprint
 - No need to create it, configure it or manage it if used within a ColdBox Application
- **Solid Core**
 - Multi-Threaded
 - Based on Java Concurrency Classes
 - Multiple Eviction Policies: LRU, LFU and FIFO
 - Memory Management & Memory Sensitive caching based on Java Soft References
 - High Load Tested
 - Fully Documented
- **Extensible & Flexible**
 - Cache Listeners for event broadcasting
 - Create your own custom eviction policies
 - Create your own cache providers
 - Create your own CacheBox object storages
 - Extensive and granular purging mechanisms (regular expressions and key snippets)
 - Cache Statistics API for creating custom cache reports
- **Highly Configurable**
 - JVM Threshold Checks
 - Object Limits
 - Ability to time expire objects
 - Eternal (singletons) and time-lived objects
 - Object purging based on object usage (Access Timeouts)
 - Fully configurable at runtime via dynamic configurations and hot-updates
- **Visually Appealing and Useful**
 - Fully featured caching monitor and commands panel
 - Complete cache performance reports

Downloading CacheBox

You can download the official download of CacheBox by going to the [ColdBox's downloads page](#). If you have ColdBox 3.0.0 already installed, don't worry, you have CacheBox already installed. The CacheBox API docs can be found also in our [main website and are also available for download](#). We highly recommend you download the latest API docs. They are definitely a high wealth of information and it will show you all the methods and properties available for operation.

Installing CacheBox

CacheBox has been designed to work either as a standalone framework or within the ColdBox Platform. So if you are building ColdBox applications, you do not have to do anything; CacheBox is already part of the platform. However, if you are using CacheBox as a standalone framework, just drop the distribution folder into your webroot:

```
/cachebox/system
```

That's it! You can also create per-application mappings or global ColdFusion mappings to `/cachebox` that points to the standalone distribution folder.

System Requirements

CacheBox has been designed to work under the following CFML Engines:

- Railo 3.1 and beyond
- Adobe ColdFusion 8 and beyond

Other Requirements: You can use CacheBox as a standalone framework in which the starting component namespace will be `/cachebox/system/`. However, if you use CacheBox within the ColdBox core, the starting component namespace will be `/coldbox/system/`. Take this into consideration for all the samples, configurations and CFC paths.

Important Note: CacheBox is a ColdFusion 8 and above framework. The Adobe ColdFusion caching providers is targeted for ColdFusion 9.0.1 and above.

Useful Resources

- [CacheBox Release Notes](#)
- <http://www.coldbox.org/wiki/Cache>
- http://www.coldbox.org/development/getting_started.html
- <http://www.oracle.com/technetwork/middleware/ehcache/ehcache-113666.html>
- <http://www.couchdb.org/>
- <http://www.ibm.com/coldfusion/ehcache/ehcache113666.html>
- <http://www.ibm.com/coldfusion/ehcache/ehcache113666.html>
- <http://www.ibm.com/coldfusion/ehcache/ehcache113666.html>
- <http://www.ibm.com/coldfusion/ehcache/ehcache113666.html>

Caching Concepts

"A cache (pronounced /kæʃ/ kash) is a component that improves performance by transparently storing data such that future requests for that data can be served faster" - Wikipedia

Caching is everywhere around us these days in all shapes and forms: query caching, data caching, page fragment caching, event caching, etc. Much like how in Forrest Gump you could do all kinds of shrimp dishes :)

In its most basic form, we use caching to accelerate and scale our processes. We might have a certain request that takes time to process due to let's say in colloquial terms "a big ass query". We can leverage caching by slapping that query up and placing it in cache so we don't have to wait for it again. Our process now only at most take 1 big hit and then can be merry on their way serving requests. Again, this is a very practical approach to caching. As you will see from this guide, caching can get very very complex and you need the right tools to be able to scale out, configure and tune your caching approaches.

When building enterprise class applications we are faced with many problems dealing with architecture, performance, scalability and so much more. One of these issues might be dealing with expensive queries, but there are also lots of other issues to consider like: expense of object creation/configuration, data retrieval, data/system availability, page output caching, etc. All of these issues can benefit from some kind of caching in order to optimize performance, availability and scalability.

Let's explore some benefits of caching:

- Reduce the amounts of data transfers between processes, network or applications
- Reduce processing time within a system or application
- Reduce I/O access operations
- Reduce database access operations
- Reduce expensive data transformations and live by a 1 hit motto

Caching Considerations

Whenever you are leveraging the power of caching, we recommend that you also take into consideration several key factors that need to be evaluated and put into practice in your applications. Below are some key factors that we recommend:

- **Thread Safety:** Take into consideration that when you cache data/objects or whatever, multiple threads will be trying to access it in your application. Make sure that you have the appropriate locking and synchronization techniques so multiple threads don't interfere with one another. This might be from simple testing procedures to make sure an element exists, to a more strict approach where you create a cache decorator that does hard blocks on cache access via named locks. Don't ever assume that what you ask from the cache actually exists.
- **Serialization:** If you will be caching to disk, database or using replication, distribution, most likely the caching engines will be serializing or converting your data and objects into byte or string format in order to persist them. Therefore, your objects must be able to support serialization and also be aware that if you are caching complex objects, those objects will be serialized alongside the target object. This is what's called an **object graph**. Where a serializer will go through the entire object graph (object/data relationships) of the target object and serialize everything. If you have circular references or references to tons of objects, serialization performance degrades and could potentially cause a heap overflow and crash your application server as it will recurse forever. ColdFusion 9 introduced the component and property attribute called **serializable**, which is a boolean attribute you can add to the `cfcomponent` and `cfproperty` tags. We recommend using it and marking components and relationships that do NOT need serialization.

```
<--- Marking a component as NOT serializable --->
```

```
<cfcomponent output="false" serializable="false">
```

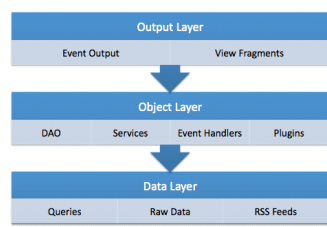
```
</cfcomponent>
```

```
<--- Marking a property as NOT serializable --->
```

```
<cfproperty name="cacheEngine" serializable="false"/>
```

Important: The default value for the `serializable` attribute is **true**.

- **Data Format:** Determine how you want to store data so it is easy to take that snapshot out of a cache and use. This can be from implementing object state/memento patterns, to leveraging serialization, to pre-defined data structures. Think ahead of HOW your data will be stored in the caches.
- **Cache Layers:** In our experience, we have seen greater improvement in performance and scalability by planning caching layers. Caching layers is what are the different layers of caching your application will provide from data to objects to event and view fragments. We recommend you analyze your cache layers and plan how they will behave. A perfect live example of caching layers can be seen in our open source ColdFusion wiki [CodexWiki](#), which you are actually using right now. CodexWiki leverages query caching, XML/Feed caching, object caching of DAO's, services, some business objects, plugins and event handlers, to view fragment caching and overall event output caching. We use it all baby!



- **Cache Limits:** There are definite benefits of demarcating limits on your cache. This involves setting up default limits for timeouts, idle timeouts and maximum number of objects within a cache. Studies have shown that leverage memory sensitive data constructs with fixed limit caching parameters can increase performance and overall JVM stability. Look at our cache resources for these studies as they are very interesting.

Cache Loading

When working with a cache engine you also have to consider how you will get data INTO the cache before you work with it. There are several approaches but I will talk about two methods of content loading:

- **Lazy Loading or Reactive Loading:** This type of loading is the most typical way to load data into a cache and it happens once data is requested from an external process. Below is a typical example using CacheBox in a service layer call:

```

/**
 * Get some blog categories from the database or cache
 */
function getCategories() {
    var cacheKey = "#big-categories#"

    // Check if data exists
    if( cache.lookup( cacheKey ) ) {
        return cache.get( cacheKey );
    }

    // Else query DB and cache
    var data = blogDAO.getCategories();
    cache.set(cacheKey, data, 120, 20);

    return data;
}
  
```

This is great for small content pieces and works well for applications. However, if you have large amounts of data that must be available at application startup or cache startup. Then we recommend looking at the next method.

- **Proactive Loading:** This approach focuses itself on the loading of resources before the application starts up so content can be loaded. EHCache for example offers cache loaders that you can create that will populate the cache on initializations. CacheBox offers the same capabilities through its event model, so you can tap into the necessary events and then carry your population and loading procedures. For example, you might tap into the `afterCacheRegistration` event in CacheBox so you might listen to when a cache engine gets created and configured so you can start populating it right at application/cache startup. In this loader you will then have the opportunity to load your data either **asynchronously** or **synchronously**. If you will be loading large amounts of data or processor intensive data, we recommend you leverage `cfthread` and do the loading asynchronously within your event interceptor.

```

/**
 * A cache listener for CacheBox
 */
component {
    /**
     * Listen for cache registrations and load data into the 'bigCache' cache ONLY!
     */
    function afterCacheRegistration( interceptData ) {
        // Get the registered cache reference from the incoming interception data
        var cache = arguments.interceptData.cache;

        // Only work on the BigCache cache
        if( cache.getName() == "BigCache" ) {

            // Talk to service, get some big data items
            dataItems = service.getBigDataItems();

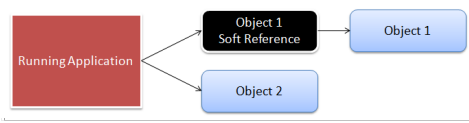
            // Cache them forever as eternal objects
            cache.setMulti( mapping=dataItems, timeout=0 );
        }
    }
}
  
```

Definitions

- **Cache Hit:** An event that occurs when an element requested from cache is found in the cache.
- **Cache Miss:** An event that occurs when an element requested from cache is NOT found in the cache.
- **Eviction:** The act of removing an element from the cache due to certain criteria algorithm that directly is connected to the state of the element.
- **Eviction Policy:** The algorithm that decides what element(s) will be evicted from a cache when full or a certain criteria has been met in the cache. ([Cache Algorithms](#))
- **LRU:** An eviction policy that discards the least recently used items first.
- **LFU:** An eviction policy that counts how often an item has been accessed and it will discard the items that have been used the least.
- **FIFO:** An eviction policy that works as a queue, first object in will be the first object out of the cache. So older staler objects are purged.
- **Cache Provider:** A concrete implementation to a caching engine within CacheBox
- **Object Store:** An object that stores cached elements and indexes them
- **Idle or Last Access Timeout:** An expiration time an element receives if it is not accessed.
- **Reap:** Usually an asynchronous event that cleans a cache from dead elements or expired elements
- **Distributed/Partitioned Cache:** A form of caching that allows the cache to span multiple servers so that it can grow in size and in capacity. Each machine contains a unique partition of the dataset. Adding machines to the cluster will increase the capacity of the cache. Both read and write access involve network transfer and serialization/deserialization.
- **Near Cache:** Each cache server contains a small local cache which is synchronized with a larger distributed/partitioned cache, optimizing read performance. There is some overhead involved with synchronizing the caches.
- **Replicated:** Each cache server contains a full copy of the elements cached
- **In Process Cache:** A cache that lives inside of the same heap as the application using it.
- **Out of Process Cache:** A cache that lives as its own process and heap in the most likely the same server as the application using it.
- **Eternal Objects:** Eternal objects are objects that will never be purged or evicted by the caching engine automatically. The only way to remove them is to recreate the cache or clear them manually.
- **Cache Topology:** A concept that refers to where data physically resides and how it is accessed in a distributed environment

Java Soft References

"In the past, developers didn't have much control over garbage collection or memory management. Java2 has changed that by introducing the java.lang.ref package. The abstract base class Reference is inherited by classes SoftReference, WeakReference and PhantomReference. SoftReferences are well suited for use in applications needing to perform caching."



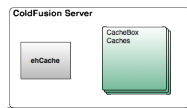
Why is this? Well, a soft reference is nothing but a wrapper class that wraps an object in memory. Whenever the JVM requires memory and runs its reference rules, it can detect these soft references and decide to purge them if it meets the garbage collector rules. If it purges the wrapped object inside of the soft reference is marked for collection, but the java soft reference itself still exists. Therefore, the programmer can determine if the object inside the reference exists or not. If it does not exist, it means the object was garbage collected and I have no more object. I won't go into the implementation semantics of the cache, just theory. Please visit the references to understand more of how the ColdBox cache was built. In summary, soft references are what determine if an object is still available or not. CacheBox cache can then run maintenance on itself and clean out all of its references for you if you are using the `ConcurrentSoftReferenceStore` object store.

Cache Topologies

There are several cache topologies that you can benefit from and it all depends on your business requirements and scalability requirements. I will only mention four distinct approaches for caching. CacheBox at this point can support all caching topologies as the underlying caches can support them. However, at this point in time, the CacheBox caching provider only supports a single instance in-process approach. However, we are working on a CacheBox server to provide out-of-process caching support and a replicated/synchronized version via [ColdBox Message](#).

Single Instance/In-Process

A single instance cache is an in-process cache that can be used within the same JVM heap as your application. In the ColdFusion world, this cache topology is what ColdFusion 9 actually offers: An instance of `cbCache` that spawns the entire JVM of the running ColdFusion server. There are pros and cons to each topology and you must evaluate the risks and benefits involved with each approach.



Definitely having a single instance approach is the easiest to setup and use. However, if you need to add more ColdFusion instances or you need to cluster your system, single instance will not be of much use anymore as servers will not be synchronized with the same cached data. CacheBox also allows you to create as many instances of itself as you need and also as many CacheBox caches as you need. This allows you great flexibility to create and configure multiple instance caches in a single ColdFusion server instance. This way if you are in shared hosting you do not have to worry about the underlying cache configuration (which is only 1), but you can configure your caching engines for your application ONLY! This is of great benefit and greater flexibility than dealing with a single cache instance on the ColdFusion server.

Benefits:

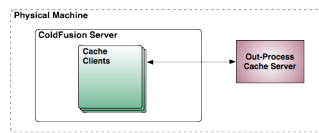
- Fast access to data as it leaves in the same processes
- Easy to setup and configure
- One easy configuration if you are using ColdFusion 9 or Railo
- You can also have multiple CacheBox caching providers running on one machine (Greater Flexibility)
- You can also have multiple CacheBox instances running on one machine or have a shared scope instance (Greater Flexibility)
- Each of your applications, whether ColdBox or not, can leverage CacheBox and create and configure caches (Greater Flexibility)

Cons:

- Shared resources in a single application server
- Shared JVM heap size, thus available RAM
- Limited scalability
- Not fault tolerant

Single Instance/Out-Process

A single instance can be an out-process cache that leaves on its own JVM typically in the same machine as the application server. This approach has also its benefits and cons. A typical example of an out of process cache can be using an instance of `CacheDB` for storage of your cache components and your applications talk to the cache via REST.



Benefits:

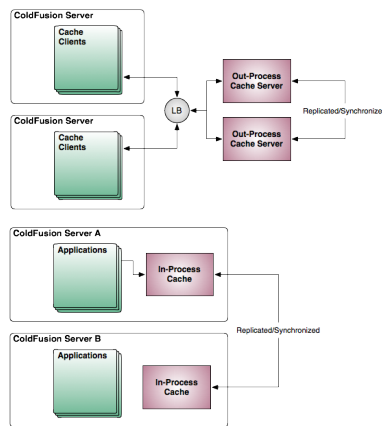
- Still access is fast as it is in the same machine
- Easy to setup and configure
- Might require a windows or *nix service so the cache engine starts up with the machine
- Can leverage its own JVM heap, memory, GC, etc and have more granularities.
- Out of process cache servers can be clustered also to provide you with better redundancy. However, once you start clustering them, each of those servers will need a way to replicate and synchronize each other.

Cons:

- Still shares resources in the server
- Limited scalability
- Needs startup scripts
- Needs a client of some sort to be installed in the application server so it can function and a protocol to talk to it: RMI, JMS, SOAP, REST, etc.
- Not fault tolerant

Replicated

A replicated cache can exist in multiple application server machines and their contents will be replicated, meaning all caches will contain all the data elements. This type of topology is beneficial as it allows for all members in the application cluster to have availability to its elements. However, there are also some drawbacks due to the amount of chatter and synchronization that must exist in order for all data or cache nodes to have the same data fresh. This approach will provide better scalability and redundancy as you are not limited to one single point of failure. This replicated cache can be in-process or out-process depending on the caching engine you are using. Our recommendation is to leverage `cbCache` and CacheBox Replicated edition (once it comes out).



Benefits:

This interface is the contract for statistics reporting. This is how CacheBox can produce gorgeous charting and reporting based on the implementation of this class.

IOjectStore

class : *coldbox.system.cache.store.IObjectStore*

A CacheBox cache provider uses the concept of an object store in order to, well, store its cached objects. This can be anything as we have created an interface that anybody can implement and create their own object storages. Just be aware that the CacheBox provider's eviction policies use certain methods for reporting and evictions that **MUST** be implemented.

IEvictionPolicy

class : *coldbox.system.cache.policies.IEvictionPolicy*

This is the contract for creating eviction policies for usage by the CacheBox cache provider.

AbstractEvictionPolicy

class : *coldbox.system.cache.policies.AbstractEvictionPolicy*

This is the abstract class that gives the magic to eviction policies under the CacheBox cache provider. It communicates with the internal object stores and asks for specific metadata in specific sorter orders so the policy can execute its algorithm. Our current algorithms are: *LRU*, *LFU*, and *FIFO*. However, you can very easily implement your own.

IColdboxApplicationCache

class : *coldbox.system.cache.IColdboxApplicationCache*

This interface extends from the original cache provider interface and adds certain methods that **MUST** be implemented so the cache can work for ColdBox applications. This includes functionality to do event/view caching, clearing/expiration by certain regular expressions and reporting. So if you want to create your own cache implementation for ColdBox applications, then it must adhere to this contract.

Creating CacheBox

CacheBox is the core framework you need to instantiate in order to work with caching in your application. You can instantiate it with a *CacheBoxConfig* object that will hold all of your caching configurations. To configure the library you can either do it programmatically or via an xml file, the choice is yours. After the library is instantiated and configured you can ask from it a named cache provider, or interact with the caches. By default, CacheBox can be started with no configuration and it will configure itself with the default configuration that is shipped with CacheBox that can be found in the following location:

```
/coldbox/system/cache/config/DefaultConfiguration.cfc
or
/cachebox/system/cache/config/DefaultConfiguration.cfc
```

The default configuration is the following:

```
function configure(){
    // The CacheBox configuration structure DSL
    cacheBox = {
        // LogBox configuration file
        logBoxConfig "coldbox.system.cache.config.LogBox"

        // Scope registration, automatically register the cachebox factory instance on any CF scope
        // By default it registers itself on application scope
        scopeRegistration = {
            enabled true
            scope "application" // valid CF scope
            key "cacheBox"
        },

        // The defaultCache has an implicit name of "default" which is a reserved cache name
        // It also has a default provider of cachebox which cannot be changed.
        // All timeouts are in minutes
        // Please note that each object store could have more configuration properties
        defaultCache = {
            objectDefaultTimeout = 60,
            objectDefaultLastAccessTimeout = 30,
            useLastAccessTimeout true,
            reapFrequency = 2,
            freeMemoryPercentageThreshold = 0,
            evictionPolicy "LRU",
            evictCount = 1,
            maxObjects = 200,
            // Our default store is the concurrent soft reference
            objectStore "concurrentSoftReferenceStore"
            // This switches the internal provider from normal cacheBox to coldbox enabled cachebox
            coldboxEnabled false
        },

        // Register all the custom named caches you like here
        caches = {},
        // Register all event listeners here, they are created in the specified order
        listeners = []
    };
}
```

If you are using CacheBox within a ColdBox application you do not need to worry about creating or even configuring the framework, as ColdBox ships already with a default configuration for ColdBox applications. However, you also have full control of configuring CacheBox at your application level (Please see the coldbox application configuration section). The default configuration for a ColdBox application can be found in the following location:

```
/coldbox/system/web/config/CacheBox.cfc
```

In summary, CacheBox has two modes of operation:

- Standalone Framework
- ColdBox Application

If you have downloaded CacheBox as a standalone framework, then please make sure you use the correct namespace path (*cachebox.system*). Also, once you create CacheBox make sure that you persist it somewhere, either in Application scope or any other scope or a dependency injection context ([WiringBox](#)). ColdBox application users already have an instance of CacheBox created for you in every application and it is stored in the main application controller and can be retrieved via the following function:

```
// Get a reference to CacheBox
controller.getCacheBox();

// Get a reference to the default cache provider
controller.getColdboxOCM() or getColdboxOCM()

// Get a reference to a named cache provider
controller.getColdboxOCM@platform or getColdboxOCM@platform
```

Note: Most of the examples shown in this documentation refer the default framework namespace of *coldbox.system*. However, if you are using CacheBox as a standalone framework, then the namespace to use is *cachebox.system*. Happy Coding!

Creating CacheBox Examples

Let's start with the simplest example of them all which is just creating CacheBox with the default configuration. The rest of the samples show different configuration options:

```
import cachebox.system.cache.*;

// create cachebox
cachebox = new CacheFactory();
// get the default cache
cache = cachebox.getDefaultCache();
// set a value in cache, with 60 minute timeout and 20 minute idle timeout.
cache.set("MyValue", {name="Luis Majano", awesome=true, cacheWeird=true}, 60,20);

Here are some more configuration samples:

// Create CacheBox with default configuration
cacheBox createObjectComponent("cachebox.system.cache.CacheFactory");

// Create the config object with an XML configuration file
config createObjectComponent("cachebox.system.cache.config.CacheBoxConfig", expandPath("cachebox.xml"));
// Create CacheBox instance
cacheBox createObjectComponent("cachebox.system.cache.CacheFactory", config);

// Create the config object as a CacheBox DSL Simple CFC
dataCFC createObjectComponent("MyCacheBoxConfig");
config createObjectComponent("cachebox.system.cache.config.CacheBoxConfig", CFCConfig=dataCFC);
// Create CacheBox instance
cacheBox createObjectComponent("cachebox.system.cache.CacheFactory", config);

// Create the config object as a CacheBox DSL Simple CFC path only
config createObjectComponent("cachebox.system.cache.config.CacheBoxConfig", CFCConfigPath="CacheBoxConfig");
// Create CacheBox instance
cacheBox createObjectComponent("cachebox.system.cache.CacheFactory", config);

// Create CacheBoxConfig, call configuration methods on it and then create the factory
config createObjectComponent("cachebox.system.cache.config.CacheBoxConfig");
// Configure programmatically: You can chain methods, woo!
config.scopeRegistration("applicationDefaultObjectStore", "maxObject=200");
// Create CacheBox instance
cacheBox createObjectComponent("cachebox.system.cache.CacheFactory", config);
```

Common CacheFactory Methods

Here is a list of some of the most common methods you can use to interact with CacheBox. For full details, please view the [online API Docs](#).

- **addCache(any<ICacheProvider> cache)**

Register a new instantiated cache with this cache factory

- **addDefaultCache(string name)**

Add a default named cache to our registry, create it, config it, register it and return it of type: cachebox

- **clearAll()**

Clears all the elements in all the registered caches without de-registrations

- **configure(CacheBoxConfig config)**

Configure the cache factory for operation, called by the init()

- **expireAll()**

Expires all the elements in all the registered caches without de-registrations

- **getCache(string name)**

Get a reference to a registered cache in this factory

- **getDefaultCache()**

Get the default cache provider of type cachebox

- `reapAll()`

A nice way to call reap on all registered caches

- `removeCache(string name)`

Try to remove a named cache from this factory

- `replaceCache(any<ICacheProvider> cache, any<ICacheProvider> decoratedCache)`

Replace a registered named cache with a new decorated cache of the same name

- `shutdown()`

Recursively sends shutdown commands to all registered caches and cleans up in preparation for shutdown

Common Operation Examples

```
// Add another default cache type but with the name FunkyCache
funkyCache = cachebox.addDefaultCacheCache;
// Add some elements to Funky Cache
funkyCache.setEntry('how()120');

// Get a reference to a named cache
cfCache = cachebox.getCacheCache;

// Get a reference to the default named cache
cache = cachebox.getDefaultCache();

// Remove our funky cache no longer needed
cachebox.removeCacheFunkyCache;

// Create a new cache to replace a cache, the MyFunkyFunkyCache implements ICacheProvider
newCache = new MyFunkyFunkyCache({maxObjects=200,timeout=30});
// replace the CFCache with this one
cachebox.replaceCacheCache, newCache );

// Add a new cache to cachebox programmatically
newCache = new MyFunkyFunkyCache({maxObjects=200,timeout=30});
cachebox.addCache( newCache );

// Send a shutdown command to cachebox
cachebox.shutdown();
```

Important: Remember that some of the CacheBox methods announce events. So please see our event model section to see what kind of events you can listen to when working with CacheBox.

Cache Cleanup/Reaping

If you are using CacheBox within a ColdBox application, ColdBox is in charge of emitting reaping or cleanup requests to the CacheBox factory. However, if you are using the standalone version of the cache, then you will need to emit this command yourself in the desired execution point. We would suggest to do this on every request. This does not mean that the request executes every time, it means that the trigger is made and if the reaping time has been elapsed, then the caches clean themselves (CacheBox Providers Only):

```
cachebox.reapAll();
```

That's it! Just call on the `reapAll()` method and this will make sure that your caches clean themselves according to the reaping frequency selected in your configuration file.

Important: Reaping frequency and cleanup only applies to the CacheBox Provider.

CacheBox Configuration

Let's delve deeper into the rabbit hole and explore how to configure this bad boy. There are three approaches to configuring CacheBox: two programmatically and one via XML. Our preference is the programmatic approach via a simple data CFC that can encapsulate the CacheBox configuration in one single component. However, at the end of the day all configuration approaches will use the `CacheBoxConfig` object's methods in some manner. This is the CFC that you will use to configure CacheBox and its constructor can be called with the following optional arguments which determines your configuration approach:

- **none** - Means you will be using the CFCs methods for configuration
- **XMLConfig** - The absolute path to the XML configuration file
- **CFConfig** - The object reference to the simple data CFC that contains the CacheBox DSL
- **CFConfigPath** - The instantiation path of the simple data CFC that contains the CacheBox DSL

No matter what configuration you decide to use, you will always have to instantiate CacheBox with a `CacheBoxConfig` object of type: `coldbox.system.cache.config.CacheBoxConfig`. However, you have the option of either working with this CFC directly or creating a more portable configuration. This portable configuration we denote as a simple data CFC that contains the CacheBox configuration data that represents our CacheBox DSL (Domain Specific Language). This DSL is exactly the same if you are using CacheBox standalone or in any ColdBox application, so it definitely has its benefits.

Tip: Please note that you have full access to the running CacheBox's configuration object or even a specific cache's configuration structure. Therefore, you can easily change the configuration settings for a cache or CacheBox dynamically at runtime.

CacheBox DSL

In order to create a simple data CFC, just create a new CFC with one required method on it called `configure()`. This is where you will define the caching configuration data structure in a structure called `cacheBox` in the `variables` scope:

```
**
* A CacheBox configuration data object
*/
component{
    function configure(){
        cacheBox = {
        };
    }
}
```

The `cacheBox` structure can be configured with the following keys:

Key	Type	Required	Default	Description
logBoxConfig	string	false	<code>coldbox.system.cache.config.LogBox</code>	The instantiation or location of a LogBox configuration file. This is only for standalone operation.
scopeRegistration	struct	false	<code>{enabled=true,scope=application,key=cacheBox}</code>	A structure that enables scope registration of the CacheBox factory in either <i>server</i> , <i>cluster</i> , <i>application</i> or <i>session</i> scope.
defaultCache	struct	true	---	The configuration of the default cache which will have an implicit name of default which is a reserved cache name. It also has a default provider of CacheBox which cannot be changed.
caches	struct	false	{}	A structure where you can create more named caches for usage in your CacheBox factory.
listeners	array	false	[]	An array that will hold all the listeners you want to configure at startup time for your CacheBox instance. If you are running CacheBox within a ColdBox application, this item is not necessary as you can register them via the main ColdBox interceptors section.

LogBoxConfig

The `LogBoxConfig` element is used only in standalone mode of operation and tells CacheBox what configuration file to load into `LogBox`. `LogBox` is an enterprise ColdFusion logging library that is part of the ColdBox Platform. This way, you have granular control of how CacheBox and its registered caches will be producing logging and debugging data. If you do not specify a location for a custom `LogBox` configuration file, then CacheBox will instantiate `LogBox` with its own default configuration file located at:

```
coldbox.system.cache.config.LogBox.cfc
OR
cachebox.system.cache.config.LogBox.cfc
```

Example:

```
// The CacheBox configuration structure DSL
cacheBox = {
    logBoxConfig = "myapp.config.MyLogBoxConfig"
};
```

Scope Registration

CacheBox has a nifty little feature that enables itself to leech onto a specific scope in memory. It basically can auto-register itself in your environment with no further need for persistence or just to expand its location. By default CacheBox registers itself in **application** scope with a key of `cacheBox`, but you have complete control on how the scope registration should work.

Customizable Keys

Key	Type	Required	Default	Description
enabled	boolean	false	true	Enable scope registration
scope	string	false	<code>application</code>	The ColdFusion scope to persist on
key	string	false	<code>cacheBox</code>	The name of the key in the ColdFusion scope to persist on

Example:

```
// The CacheBox configuration structure DSL
cacheBox = {
    scopeRegistration = {
        enabled = true
        scope = "application"
        key = "cacheBox"
    }
};
```

Default Cache

The `defaultCache` element is in itself a structure of configuration data for the **default** cache provider of type `coldbox.system.cache.providers.CacheBoxProvider`. This is a reserved cache name and it is **MANDATORY** to declare and its name or type cannot be changed. However, if you are using CacheBox within a ColdBox application, the provider will switch itself to `coldbox.system.cache.providers.CacheBoxColdBoxProvider` by using the `coldboxEnabled` key. So let's see the configuration keys for our default cache.

Default Cache Properties:

Key	Type	Required	Default	Description
ObjectDefaultTimeout	numeric	false	60	The default lifespan of an object in minutes
ObjectDefaultLastAccessTimeout	numeric	false	30	The default last access or idle timeout in minutes
UseLastAccessTimeouts	Boolean	false	true	Use or not idle timeouts
ReapFrequency	numeric	false	2	The delay in minutes to produce a cache reap (Not guaranteed)

FreeMemoryPercentageThreshold	numeric	false	0	The numerical percentage threshold of free JVM memory to have available before caching. If the JVM free memory falls below this setting, the cache will run the eviction policies in order to cache new objects. (0=Unlimited)
MaxObjects	numeric	false	200	The maximum number of objects for the cache
EvictionPolicy	string or path	false	LRU	The eviction policy algorithm class to use. ColdBox ships with <ul style="list-style-type: none"> ● LRU (Least Frequently Used) ● LRU (Least Recently Used) ● FIFO (First In First Out) You can also build your own and pass the instantiation path in this setting
EvictCount	numeric	false	1	The number of objects to evict once an execution of the policy is requested. You can increase this to make your evictions more aggressive
objectStore	string	false	<i>ConcurrentStore</i>	The object store to use for caching objects. ColdBox ships with the following object stores: <ul style="list-style-type: none"> ● ConcurrentStore - Uses concurrent hashmaps for increased performance ● ConcurrentSoftReferenceStore - Concurrent hashmaps plus java soft references for JVM memory sensitivity ● DiskStore - Uses a physical disk location for caching (Uses java serialization for complex objects) ● JDBCStore - Uses a JDBC datasource for caching (Uses java serialization for complex objects) You can also build your own and pass the instantiation path in this setting.
ColdboxEnabled	Boolean	false	false	A flag that switches on/off the usage of either a plain vanilla CacheBox provider or a ColdBox enhanced provider. This must be true when used within a ColdBox application and it applies for the default cache ONLY.

Important : Please note that each object store could have more configuration properties that you will need to add to the configuration structure. To find out about these extra configuration properties, please see the Object Store's section.

Example:

```
// The CacheBox configuration structure DSL
cacheBox = {
// Please note that each object store could have more configuration properties
defaultCache = {
objectDefaultTimeout = 60,
objectDefaultLastAccessTimeout = 30,
useLastAccessTimeout = true,
respFrequency = 2,
freeMemoryPercentageThreshold = 0,
evictionPolicy = "LRU",
evictCount = 1,
maxObjects = 200,
// Our default store is the concurrent soft reference
objectStore = "ConcurrentSoftReferenceStore"
// This switches the internal provider from normal cacheBox to coldbox enabled cachebox
coldboxEnabled = false
}
}
```

Caches

The *caches* element is in itself a structure of configuration data for aggregating named cache providers of ANY type. The key is the unique name of the cache to aggregate under this CacheBox instance. The value should be a structure with the following keys:

Key	Type	Required	Default	Description
provider	string	true	---	The instantiation path of the cache that must implement our <i>ICacheProvider</i> interface.
properties	struct	false	{}	A structure of name-value pairs of configuration data for the cache to declare.

Example:

```
// Register all the custom named caches you like here
caches = {
sampleCache1 = {
provider = "coldbox.system.cache.providers.CacheBoxProvider"
properties = {
objectDefaultTimeout = 60,
objectDefaultLastAccessTimeout = 30,
useLastAccessTimeout = true,
respFrequency = 2,
evictionPolicy = "LRU",
evictCount = 1,
maxObjects = 200,
objectStore = "ConcurrentSoftReferenceStore"
}
}
sampleCache2 = {
provider = "coldbox.system.cache.providers.CFColdboxProvider"
}
}
```

Listeners

This is an array of structures that you will use to define cache listeners. This is exactly the same as if you would be declaring ColdBox [Interceptors](#), so order is very important. The order of declaration is the order that they are registered into their specific listening events and also the order in which they fire. The keys to declare for each structure are:

Key	Type	Required	Default	Description
class	instantiation path	true	---	The instantiation path of the listener object to register
name	string	false	listLast(class, ".")	The unique name used for this listener for registration purposes. If no name is provided, then we will use the last part of the instantiation path, which is usually the filename.
properties	struct	false	{}	A structure of name-value pairs of configuration data for the listener declared.

CacheBox Config Object

The main CacheBox DSL is a great way to configure CacheBox as it is very portable. However, you can also configure CacheBox by calling methods on the *CacheBoxConfig* object, which is exactly what we do when we read the CacheBox DSL. Below are the main methods used for configuring CacheBox which match exactly to the DSL items, so please refer to the DSL items for definitions and settings. Also, for all the methods in this object, check out the [API Docs](#).

● **init([string XMLConfig=], [any CFCConfig], [string CFCConfigPath])**

The constructor

● **cache(string name, [string provider=], [struct properties=])**

Add a new cache configuration

● **defaultCache([numeric objectDefaultTimeout=], [numeric objectDefaultLastAccessTimeout=], [numeric respFrequency=], [numeric maxObjects=], [numeric freeMemoryPercentageThreshold=], [boolean useLastAccessTimeout=], [string evictionPolicy=], [numeric evictCount=], [string objectStore=], [boolean coldboxEnabled=])**

Add a default cache configuration

● **listener(string class, [struct properties=], [string name=])**

Add a new listener configuration

● **logBoxConfig(string config)**

Set the logBox Configuration path to use

● **reset()**

Reset the entire configuration

● **scopeRegistration([boolean enabled='false'], [string scope='application'], [string key='cachebox'])**

Use to define cachebox factory scope registration

Code Examples:

```
config <createObjectcomponent> "coldbox.system.cache.config.CacheBoxConfig";
// logbox config & scope chained, yes you can chain any method
config.logBoxConfig("logBox.xml")<scopeRegistration>("application" "cacheBox");
// default cache
config.defaultCache(maxObjects=500, evictCount=5, objectDefaultTimeout=30);
// Caches
config.cache("cache", "caches_ehCacheProvider", "configFiles/cache.xml");
config.cache("template", "coldbox.system.cache.providers.CacheBoxProvider");
// Listeners
config.listener("app.model.MyListener", "MyListener");
```

XML Configuration

You can also configure CacheBox with an XML file. All you need to do is create a CacheBox XML file and instantiate the config object with the location of such config file:

```
config <createObjectcomponent> "coldbox.system.cache.config.CacheBoxConfig" <expandPath>("config/cacheBox.xml")
```

However, if you have your XML in a variable already, maybe you read it from a database or other location, you can still use it by calling the config object's *parseAndLoad()* method.

```
//Get the xml document from somewhere
xmlDoc = dbService().getCacheBoxConfig();
//create the CacheBox config object
config <createObjectcomponent> "coldbox.system.cache.config.CacheBoxConfig";
config.parseAndLoad(xmlDoc);
```

Sample *cacheBox.xml* file:

```
<?xml version="1.0" encoding="UTF-8"?>
<CacheBox xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.coldbox.org/schema/CacheBoxConfig1.0.xsd">
<< -> The LogBox Configuration File to Use << ->
<LogBoxConfig> "coldbox.system.cache.config.LogBoxConfig"
<< -> <Scope Registration << ->
<ScopeRegistration enabled="true" scope="server" key="cachebox1"/>
<< -> <The defaultCache has an implicit name 'default' which is a reserved cache name
It also has a default provider of cachebox which cannot be changed.
All timeouts are in minutes << ->
<DefaultConfiguration
objectDefaultTimeout=60
objectDefaultLastAccessTimeout=30
```

```

useLastAccessTimeOutTime*
reapFrequency*
freeMemoryPercentageThreshold*
evictionPolicy*LRU*
evictCount*
maxObjects*0*
objectStore*@concurrentSoftReferenceStore*
coldboxEnabled*true*/>
</Cache>
<!-- < Register all the named caches below -->
<CacheName=SampleCache*provider=coldbox.system.cache.providers.CacheBoxProvider*
<Properties
objectDefaultTimeOutTime*
useLastAccessTimeOutTime*
reapFrequency*
evictionPolicy*LRU*
evictCount*
maxObjects*0*
objectStore*@concurrentSoftReferenceStore*
</Cache>
<CacheName=sampleCache*provider=coldbox.system.cache.providers.CacheBoxProvider*
<PropertiesmaxObjects*0*
evictionPolicy*LRU*/>
</Cache>
<!-- < Register cache listeners -->
<CacheListeners>
<ListenerClass=test.path.ListenerName=SampleListener*
<PropertyNames>*/Property*
</Listener*
</CacheListeners*
</CacheBox>

```

As you can see the XML elements translate to the CacheBox DSL elements, so please refer to them for their meaning. The root element must be `<CacheBox>` with the following child elements:

- `<LogBoxConfig>`: The location of the LogBox configuration file.
- `<ScopeRegistration>`: The scope registration for the factory
 - `@scope`: The ColdFusion scope
 - `@key`: The key in the scope
 - `@enabled`: True/False to enable registration
- `<DefaultConfiguration>`: The *default* reserved cache configuration. All attributes match the CacheBox DSL default cache properties.
 - `@objectDefaultTimeOut`
 - `@objectDefaultLastAccessTimeout`
 - `@useLastAccessTimeOut`
 - `@reapFrequency`
 - `@freeMemoryPercentageThreshold`
 - `@evictionPolicy`
 - `@evictCount`
 - `@maxObjects`
 - `@objectStore`
 - `@coldboxEnabled`
- `<Cache>`: A cache configuration, you can have as many as you like of these elements. They also match exactly to the `caches` element in the CacheBox DSL.
 - `@name`: Mandatory unique name for the cache.
 - `@provider`: The class of the provider
 - `@properties`: Optional child element that you can use to configure the name-value pairs of configuration properties for the cache. Each property is an attribute of this XML element.
- `<CacheListeners>`: A container element for all the cache listeners to register
 - `<Listeners>`: A listener declaration
 - `@class`: The instantiation class
 - `@name`: The unique name of the listener, not mandatory, defaults to the filename in the `class` attribute
 - `<Property>`: 0 or more configuration properties for the listener
 - `@name`: Name of the property
 - `XMLValue`: The value of the XML element is the value of the property.

To make this XML configuration easier, CacheBox comes with its own XML Schema that can be found in the following location:

```

/coldbox/system/cache/config/CacheBoxConfig.xsd
or
/coldbox/system/cache/config/CacheBoxConfig.xsd
or
http://www.coldbox.org/schema/CacheBoxConfig_1.0.xsd

```

You can also add the location to your XML header so your IDE can give you introspection and validation:

```

<?xml version="1.0" encoding="UTF-8"?>
<CacheBox xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.coldbox.org/schema/CacheBoxConfig_1.0.xsd"
</CacheBox>

```

Note: Please note the version identifier in the online schema, make sure you have the correct version when using within an IDE.

ColdBox Application Configuration

Every ColdBox application can use CacheBox without any extra creation or configuration. By default, every ColdBox application is configured with a CacheBox instance using the configuration file found in:

```
/coldbox/system/web/config/CacheBox
```

The contents of this configuration CFC is the following:

```

function configure() {
    // The CacheBox configuration structure DSL
    cacheBox = {
        // LogBox config already in coldbox app, not needed
        // logBoxConfig = "coldbox.system.web.config.LogBox",
        // The defaultCache has an implicit name "default" which is a reserved cache name
        // It also has a default provider of cachebox which cannot be changed.
        // All timeouts are in minutes
        defaultCache = {
            objectDefaultTimeOut = 120, // 2 hours default
            objectDefaultLastAccessTimeout = 30, // 30 minutes idle time
            useLastAccessTimeOutTime = true,
            reapFrequency = 2,
            freeMemoryPercentageThreshold = 0,
            evictionPolicy = LRU,
            evictCount = 2,
            maxObjects = 300,
            objectStore = @concurrentStore // guaranteed objects
            coldboxEnabled = true
        },
        // Register all the custom named caches you like here
        caches = {
            // Named cache for all coldbox event and view template caching
            template = {
                provider = "coldbox.system.cache.providers.CacheBoxColdBoxProvider"
                properties = {
                    objectDefaultTimeOut = 120,
                    objectDefaultLastAccessTimeout = 30,
                    useLastAccessTimeOutTime = true,
                    reapFrequency = 2,
                    freeMemoryPercentageThreshold = 0,
                    evictionPolicy = LRU,
                    evictCount = 2,
                    maxObjects = 300,
                    objectStore = @concurrentSoftReferenceStore // memory sensitive
                }
            }
        }
    }
}

```

This configuration file configures CacheBox with two caches: *default* and *template*. The *default* cache is used for persistence of handlers, plugins, interceptors, and model objects. The *template* cache is used for event caching and view fragment caching. It is also important to note that the *template* cache uses the *ConcurrentSoftReferenceStore* for its objects. This means that even view caching is subject to available memory via the JVM, a memory sensitive cache. This is essential as you could have lots of events being cached and you want to be considerate with memory concerns. It is also limited to 300 objects.

Now, this is all nice and dandy, but what if I want to configure CacheBox "MY WAY!!!"? Well, don't shout, we are just getting there. There are several ways you can configure CacheBox from within your applications, so choose wisely.

Programmatic Configuration

ColdBox 3.0.0 and above allows for a programmatic approach via the ColdBox configuration object. So let's look at how the framework loader looks at your configuration for CacheBox configuration details:

- Is there a `CacheBox` DSL variable defined in the configuration structure?
 - **False:**
 - Does a `CacheBox.cfc` exist in the application's config folder?
 - **True:** Use that CFC by convention to configure CacheBox
 - **False:** Configure CacheBox with default framework settings found in `/coldbox/system/web/config/CacheBox.cfc`
 - **True:**
 - Have you defined a `configFile` key in the `cacheBox` DSL structure?
 - **True:** Then use that value to pass into the configuration object so it can load CacheBox using that configuration file (xml or CFC)
 - **False:** The configuration data (CacheBox DSL) is going to be defined inline here so use that structure for configuration

Note: The configuration DSL is exactly the same as you have seen in before with the only distinction that you can add a `configFile` key that can point to an external configuration file (XML or CFC)

```

cacheBox = {
    configFile = "myPath.config.CacheBoxConfig"
},

```

That's it folks! You can either write inline CacheBox DSL configuration, use by convention a `CacheBox.cfc` data CFC or use a `configFile` approach for external file loading. Pick your poison!

XML

The XML approach uses exactly the same configuration elements as the normal XML configuration file but with one extra element: `<ConfigFile>`. This serves the same purpose as in the programmatic approach, where you have defined the CacheBox configuration somewhere and you are pointing to it via this setting:

```

<CacheBox>
<ConfigFile shared.path.LogBoxConfig\ConfigFile*
</CacheBox>

```

Please also note that the application loader follows the same approach as in the programmatic configuration:

- Is there a `<CacheBox>` element defined in the configuration?
 - **False:**
 - Does a `CacheBox.cfc` exist in the application's config folder?
 - **True:** Use that CFC by convention to configure CacheBox

- **False:** Configure CacheBox with default framework settings found in `/coldbox/system/web/config/CacheBox`
- **True:**
 - Have you defined a `<ConfigFile>` key in the `cacheBox` XML structure?
 - **True:** Then use that value to pass into the configuration object so it can load CacheBox using that configuration file (xml or CFC)
 - **False:** The configuration data (XML) is going to be defined inline here so use that XML for configuration

Cache Providers

We have shipped CacheBox with several CacheBox providers that you can use in your applications. There are basically two modes of operation for a CacheBox provider and it is all determined by the interface they implement:

1. **ICacheProvider**: A standalone cache provider
2. **IColdboxApplicationCache**: A cache provider for usage by the ColdBox Framework

The following are the shipped providers:

Provider	ColdBox Enabled	Reporting Enabled	Description
CacheBoxProvider	false	true	Our very own CacheBox caching engine
CacheBoxColdBoxProvider	true	true	Our CacheBox caching engine prepared for ColdBox application usage
CFProvider	false	true	A ColdFusion 9.0.1 and above implementation
CFColdBoxProvider	true	true	A ColdBox enhanced version of our ColdFusion 9.0.1 cache provider
RailoProvider	false	true	A ColdBox enhanced version of our Railo cache provider
RailoColdBoxProvider	true	true	A ColdBox enhanced version of our Railo cache provider
MockProvider	true	false	A ColdBox enhanced cache provider that can be used for mocking or testing

Each provider has the shared functionality provided by the **ICacheProvider** and **IColdboxApplicationCache** interfaces, so I encourage you to look at the [CFC API Docs](#) for an in-depth view of their API. Also, please note that each cache provider implementation has also some extra methods and functionality according to their implementation, so please check out the API docs for each provider.

CF Providers

Our *CFColdboxProvider* is an implementation specifically written for Adobe ColdFusion 9.0.1 and beyond. This provider leverages the [EHCACHE](#) engine within ColdFusion 9 and extends the native ColdFusion capabilities by talking to the EHCACHE sessions natively via Java. In this manner we are able to do things like:

- Get extended cached object metadata
- Get overall cache statistics
- Talk to terracotta classes
- Do quiet operations on get, clear, and put
- Check if an element is expired
- Do reporting and content reporting
- Much more

Properties

Each CacheBox provider can have its own set of configuration properties it needs for operation. Our CF providers also have some:

Property	Type	Required	Default	Description
<code>cacheName</code>	string	false	<i>object</i>	The named cache to talk to via ColdFusion cache operations. By default we talk to the default ColdFusion object cache.

Note: Please note that you can configure more than 1 *CFColdboxProvider* cache engine in your applications that can talk to more than one referenced ColdFusion (EHCACHE) custom cache.

Railo Providers

Our *RailoProvider* is an implementation specifically written for Railo.

Properties

Property	Type	Required	Default	Description
<code>cacheName</code>	string	false	<i>object</i>	The named cache to talk to via Railo cache operations. By default we talk to the default Railo object cache.

Mock Provider

This provider can be used for mocking capabilities in your unit tests or any kind of test you are performing. It keeps a simple localized cache inside of a local structure. It should definitely not used for any type of caching at all.

Properties

- *None*

CacheBox Provider

The CacheBox provider is our very own enterprise cache implementation that has been part of the core since its initial baby step versions. Our caching engine has matured over the years and it has been proven effective, fast, scalable and very user friendly. In this release, we took all our gained knowledge and improved it considerably. We decoupled our object storages and created a way to create object storages for caching purposes via our *IObjectStore* interface. This allows us to get creative and be able to extend the caching provider into different and creative object stores. We fine tuned it for high availability, made it much more scalable, and expanded event caching to support multi-domain hosting and so much more. Here are the two implementations for our CacheBox provider:

1. **CacheBoxProvider**: Our base CacheBox provider caching engine
2. **CacheBoxColdBoxProvider**: A subclass of *CacheBoxProvider* that enables caching operations for ColdBox applications

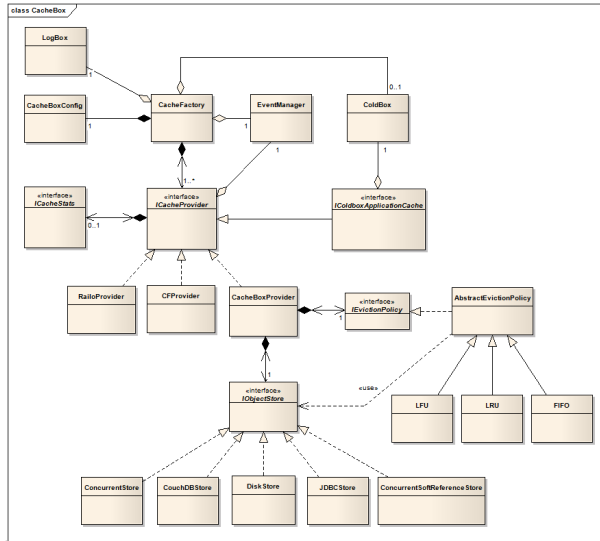
Properties

Key	Type	Required	Default	Description
<code>ObjectDefaultTimeout</code>	numeric	false	60	The default lifespan of an object in minutes
<code>ObjectDefaultLastAccessTimeout</code>	numeric	false	30	The default last access or idle timeout in minutes
<code>UseLastAccessTimeouts</code>	Boolean	false	true	Use or not idle timeouts
<code>ReapFrequency</code>	numeric	false	2	The delay in minutes to produce a cache reap (Not guaranteed)
<code>FreeMemoryPercentageThreshold</code>	numeric	false	0	The numerical percentage threshold of free JVM memory to have available before caching. If the JVM free memory falls below this setting, the cache will run the eviction policies in order to cache new objects. (0=Unlimited)
<code>MaxObjects</code>	numeric	false	200	The maximum number of objects for the cache
<code>EvictionPolicy</code>	string or path	false	LRU	The eviction policy algorithm class to use. ColdBox ships with <ul style="list-style-type: none"> ● LFU (Least Frequently Used) ● LRU (Least Recently Used) ● FIFO (First In First Out) You can also build your own and pass the instantiation path in this setting
<code>EvictCount</code>	numeric	false	1	The number of objects to evict once an execution of the policy is requested. You can increase this to make your evictions more aggressive
<code>objectStore</code>	string	false	<i>ConcurrentStore</i>	The object store to use for caching objects. ColdBox ships with the following object stores: <ul style="list-style-type: none"> ● ConcurrentStore - Uses concurrent hashmaps for increased performance ● ConcurrentSoftReferenceStore - Concurrent hashmaps plus java soft references for JVM memory sensitivity ● DiskStore - Uses a physical disk location for caching (Uses java serialization for complex objects) ● JDBCStore - Uses a JDBC datasource for caching (Uses java serialization for complex objects) You can also build your own and pass the instantiation path in this setting.
<code>ColdboxEnabled</code>	Boolean	false	false	A flag that switches on/off the usage of either a plain vanilla CacheBox provider or a ColdBox enhanced provider. This must be true when used within a ColdBox application and it applies for the default cache ONLY.

Please also note that each of the object stores can have extra configuration properties that you will need to set. So for that, let's delve a little deeper into object stores.

CacheBox Object Stores

CacheBox offers the capability of object stores that can be used to store cached objects. If you look at the following figure you will understand our object model for our object stores:



Each object store is composed into a CacheBox provider for usage of storing and retrieving cached objects. The design principle behind each object store is that each object store implements its own locking, serialization, reading and writing mechanisms that are fronted by a nice API via the *CacheProvider* object. Also, it is very important to note that the CacheBox eviction policies talk to the object stores in order to get ordered data structures for eviction purposes. So if you will be creating your own object stores, make sure they implement the right data methods the eviction policies can use for evictions. Also, the CacheBox provider's reaping methods use the object stores in order to get metadata out of the object stores in order to take care of cleanup, expirations and evictions. We definitely encourage you to take a look at the written object store implementations in order to learn how this works. So let's start investigating each of the object stores that are shipped with CacheBox.

Important: Please note that each object store can have extra properties that need to be set in your cache configuration file.

ConcurrentStore

The *ConcurrentStore* is an object store that uses concurrent hash maps provided by the Sun JDK. Concurrent maps are better suited for caching algorithms as they do not lock the entire map in order to access elements. They work on the concept of row locking. Therefore, the increased performance and increase in throughput is observable by using concurrency maps. Here is a quote from the Java Docs:

"However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove). Retrievals reflect the results of the most recently completed update operations holding upon their onset." - [Java Docs](#).

Properties

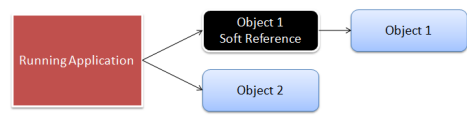
- None

You can also find some great research out there about concurrency maps:

- <http://www.ibm.com/developerworks/java/library/j-tp07233.html>
- <http://www.wheatliffe.org>

ConcurrentSoftReferenceStore

The *ConcurrentSoftReferenceStore* uses a combination of concurrency maps and java soft references. In our introductory sections we introduced java soft references and how it allows us to create placeholders in memory for objects and when the JVM needs memory it has permission to collect these placeholders and thus release memory. This is truly a great combination to have and ColdBox leverages this for its template, event and view fragment caching.



Just remember that when you use this object storage, the time spans are never guaranteed as the JVM has permission to cleanup objects when it sees fit. To get much better performance on the JVM memory heap space and a more active memory, we recommend using this object storage. Research has also indicated that a combination of memory sensitive structures in combination with limits in a cache, can increase performance and optimize its usage. So please make sure limits are set in the cache.

Properties

- None

You can also find some great research out there about concurrency maps and soft references:

- <http://www.ibm.com/developerworks/java/library/j-tp07233.html>
- <http://www.ibm.com/developerworks/java/library/j-tp01246.html>

DiskStore

The *DiskStore* as its name implies, uses a disk to store cached objects under. This can be any location that can be accessible via *efile* and *cfirectory*. Therefore, you can use Railo resources or virtual RAM locations or even Amazon S3. You can get funky my friends!

Anyways, this storage will look at the incoming target object to cache and if it is a complex object it will serialize to binary and store it on disk. If it is a simple value, it just saves it. This is a great way to provide a centralized template/view caching mechanisms as views are simple strings. You can even connect a farm of servers to talk to a centralized network drive that has centralized view/event template storage. Anyways, you can get funky!

Properties

Key	Type	Required	Default	Description
autoExpandPath	Boolean	false	true	A flag that indicates if the store should use <i>expandPath()</i> to figure out the path to store objects in.
directoryPath	string	true	---	The directory path to use to store cached objects under. This can be relative or absolute.

JDBCStore

The *JDBCStore* is a nice object storage that leverages a single database table to keep track of its objects. This is great for simple or small scale clusters that need centralized caching. It also follows suit that complex objects are serialized into the tables and simple objects are just saved. The object store can even create the database table for you if need be via its own object store custom properties.

This cache provider is a great way to use for flash variables using the ColdBox [Flash RAM](#)

Properties

Key	Type	Required	Default	Description
dsn	string	true	---	The datasource to connect to
table	string	true	---	The table we will be caching to
dsnUsername	string	false	---	The DSN username if used
dsnPassword	string	false	---	The DSN password if used
tableAutoCreate	Boolean	false	true	The object store can create the repository table for you if need be and if it does not exist.

BlackholeStore

The *BlackholeStore* is not something you would actually deploy to a production site. The *BlackholeStore* is a development tool that simulates caching but never actually caches anything. This store can be useful to help tune an application as it can provide a solid baseline for how things would behave without caching. Anything set into a cache powered by the blackhole store will vanish into a blackhole. Anytime a lookup is performed it will appear as if that object has expired. Finally, if you actually try to *get* and object you will receive a *null*.

CacheBox Eviction Policies

The *CacheBox Provider* offers several eviction policies that are used to evict objects from cache when certain situations occur within the cache environment. Again, these eviction policies are only for the CacheBox Cache provider. These eviction situations can include:

- Maximum objects reached
- JVM memory threshold reached
- Manual eviction execution

Below is a refresher on what an eviction policy means:

Eviction Policy: The algorithm that decides what element(s) will be evicted from a cache when full or a certain criteria has been met in the cache. (Cache Algorithms)

Please note that when a cache element is evicted from the cache it usually is expired first so it can be gracefully collected when the CacheBox reaping procedures occur. Therefore, you might see sometimes that the cache actually goes above the maximum objects defined, this is normal.

Important: Please note that **no** eternal objects are ever evicted from the CacheBox provider. Eternal objects live as long as the CacheBox instance lives for (most likely application scope timeout). So please take that into consideration.

ColdBox ships with the following eviction policies:

Policy	Description
LRU (Least Recently Used)	With this eviction policy, the cache discards the least recently used items first. <i>* This is the default policy</i>
LFU (Least Frequently Used)	This policy counts how often an item has been accessed and it will discard the items that have been used the least.
FIFO (First In First Out)	Just like it sounds, first one in, first one out. Great for implementing sized queues
LIFO (Last In First Out)	Just like it sounds, last one in, first one out. Great for implementing sized stacks or timed stacks

Using Your Own Policy

CacheBox is incredibly flexible and if you would like to create your own eviction policy, you can! Below are a set of easy steps on how to do this:

1. Create a simple CFC that implements the following class `coldbox.system.cache.policies.EvictionPolicy` or use our convenience abstract class and inherit from `coldbox.system.cache.policies.AbstractEvictionPolicy`. 2. Create your own `execute()` method that will evict items (We recommend looking at existing policies to get an insight on how to do this) 3. Use the policy instantiation path in your cachebox provider properties

Sample Policy

```
<cfcomponent output="false"
  hint="FIFO Eviction Policy Command"
  extends="coldbox.system.cache.policies.AbstractEvictionPolicy">
<!------- CONSTRUCTOR ----->
<!-- init -->
<cffunction name="init" output="false" access="public" returntype="FIFO" hint="Constructor">
<cfargument name="cacheProvider" type="any" required="true" hint="The associated cache provider of type: coldbox.system.cache.ICacheProvider">
<cfscript>
  super.init(arguments.cacheProvider);

  return this;
</cfscript>
</cffunction>
<!------- PUBLIC ----->
<!-- execute -->
<cffunction name="execute" output="false" access="public" returntype="void" hint="Execute the policy">
<cfscript>
  var index = #;

  // Get searchable index
  try {
    index = getAssociatedCache().getObjectStore().getIndexer().getSortedKeys(c,"asc");
  }
  // process evictions via the abstract class
  processEvictions( index );
}
catchAny e {
  getLogger().error("for sorting via store indexer #e.message# #e.detail# #e.stackTrace#."
}
</cfscript>
</cffunction>
<!------- PRIVATE ----->
</cfcomponent>
```

Process Evictions. The below code is the code used to evict objects from cache

```
<!-- processEvictions -->
<cffunction name="processEvictions" output="false" access="private" returntype="void" hint="Abstract processing of evictions">
<cfargument name="index" type="array" required="true" hint="The array of metadata keys used for processing #evictions">
<cfscript>
  var oCacheManager = getAssociatedCache();
  var indexer = oCacheManager.getObjectStore().getIndexer();
  var indexLength = arrayLen(arguments.index);
  var x = 1;
  var md = #;
  var evictCount = oCacheManager.getConfiguration().evictCount;
  var evictedCounter = 0;

  //Loop Through Metadata
  for (x=1; x<=indexLength; x=x+1){
    // verify object in indexer
    if (NOT indexer.objectExists( arguments.index[x] )) {
      continue;
    }
    md = indexer.getObjectMetadata( arguments.index[x] );
    // Evict if not already marked for eviction or an eternal object.
    if ( md.timeout > 0 AND NOT md.isExpired ){
      // Expire Object
      oCacheManager.expireKey( arguments.index[x] );
      // Record Eviction
      oCacheManager.getStats().evictionHit();
      evictedCounter++;
    }
    // Can we break or keep on evicting
    if ( evictedCounter >= evictCount ){
      break;
    }
  }
  //end for loop
</cfscript>
</cffunction>
```

Configuration File

```
defaultCache = {
  objectDefaultTimeout = 120,
  objectDefaultLastAccessTimeout = 30,
  useLastAccessTimeout = true,
  reapFrequency = 2,
  freeMemoryPercentageThreshold = 0,
  evictionPolicy="myPath.policies.MyPolicy"
  evictCount = 1,
  maxObjects = 100,
  objectStore "ConcurrentSoftReferenceStore"
  coldboxEnabled=false
}
```

That's it folks! Very easily you can create your own eviction policies and use our built in indexers to just sort the elements in whatever way you like. If not, you can always do it yourself :)

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. Of course, if you are within a ColdBox application, you get the benefit of all the potential of [ColdBox Interceptors](#) and if you are in standalone mode, well, you just get the listener and that's it. 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. So let's start investigating first the CacheBox life cycle events.

CacheBox Events

CacheBox's aggregation functionality offers a wide gamut of life cycle events that are announced at certain points in execution time. Below are the current events announced by the CacheBox `CacheFactory`. Remember, this is the `CacheFactory` and not a `CacheBox Cache Provider`.

Event	Data	Description
afterCacheRegistration	● cache : the registered cache reference	Called after a new cache provider has been registered, configured and initialized within CacheBox.
beforeCacheRemoval	● cache : the cache reference to remove	Called before a <code>removeCache()</code> operation is called on CacheBox
afterCacheRemoval	● cache : the cache name]	Called after the cache has been removed from CacheBox. You receive only the name of the cache removed.
beforeCacheReplacement	● oldCache : the cache reference to replace ● newCache : the new cache reference to replace with	Called right before a cache is replaced with another cache in CacheBox.
afterCacheFactoryConfiguration	● cacheFactory : A reference to the CacheBox factory created	Called after a CacheBox instance has been created, configured and started up. This is a great interception point for creating cache loaders or on startup scripts.
beforeCacheFactoryShutdown	● cacheFactory : A reference to the CacheBox factory created	Called right before the CacheBox instance is shutdown gracefully
afterCacheFactoryShutdown	● cacheFactory : A reference to the CacheBox factory created	Called right after the CacheBox instance has been shutdown gracefully
beforeCacheShutdown	● cache : A reference to the cache to shutdown	Called right before a specific cache provider is shutdown by CacheBox Factory
afterCacheShutdown	● cache : A reference to the cache that was shutdown	Called right after a specific cache provider has been shut down by CacheBox Factory and before its removed

CacheBox Provider Events

Each cache provider has the potential of announcing life cycle events as it implements it. If you are a cache provider author, then you can use the aggregated *EventManager* to register, process and announce events a-la-carte. So let's investigate each provider's life cycle events:

CacheBoxProvider-CacheBoxColdBoxProvider Events

Event	Data	Description
afterCacheElementUpdated	<ul style="list-style-type: none"> ● cache: the cache provider ● cacheNewObject: the new object to cache ● cacheOldObject: the object replaced 	Called via a <i>set()</i> operation when there is already the same key in the cache. Called before the replacement occurs
afterCacheElementInsert	<ul style="list-style-type: none"> ● cache: the cache provider ● cacheObject: the new object to cache ● cacheObjectKey: the key used to store the object ● cacheObjectTimeout: the timeout used ● cacheObjectLastAccessTimeout: the last access timeout used 	Called after a new cache element has been inserted into the cache
afterCacheElementRemoved	<ul style="list-style-type: none"> ● cache: the cache provider ● cacheObjectKey: the key of the removed object 	Called after a cache element has been removed from the cache
afterCacheClearAll	<ul style="list-style-type: none"> ● cache: the cache provider 	Called after a <i>clearAll()</i> has been issued on the cache
afterCacheElementExpired	<ul style="list-style-type: none"> ● cache: the cache provider ● cacheObjectKey: the key of the expired object 	Called after a cache element has been expired from the cache

CFProvider-CFColdboxProvider Events

Event	Data	Description
afterCacheElementInsert	<ul style="list-style-type: none"> ● cache: the cache provider ● cacheObject: the new object to cache ● cacheObjectKey: the key used to store the object ● cacheObjectTimeout: the timeout used ● cacheObjectLastAccessTimeout: the last access timeout used 	Called after a new cache element has been inserted into the cache
afterCacheElementRemoved	<ul style="list-style-type: none"> ● cache: the cache provider ● cacheObjectKey: the key of the removed object 	Called after a cache element has been removed from the cache
afterCacheClearAll	<ul style="list-style-type: none"> ● cache: the cache provider 	Called after a <i>clearAll()</i> has been issued on the cache

Cache Listeners

We have already seen in our previous section all the events that are announced by CacheBox and its providers, but how do we listen? There are two ways to build CacheBox listeners because there are two modes of operations, but the core is the same. Listeners are simple CFCs that must create methods that match the name of the event they want to listen in. If you are running CacheBox within a ColdBox application, listeners are [interceptors](#) and you declare them and register them exactly the same way that you do with normal interceptors. Also, these methods can take up to two parameters depending on your mode of operation (standalone or coldbox). The one main difference between pure CacheBox listeners and ColdBox interceptors are that the *configure* method for the standalone CacheBox is different. Please see samples.

Argument	Type	Execution Mode	Description
event	<i>coldbox.system.web.context.RequestContext</i>	coldbox	The request context of the running request
interceptData	struct	standalone-coldbox	The data structure passed in the event

So let's say that we want to listen on the **beforeCacheFactoryShutdown** and on the **afterCacheElementRemoved** event in our listener.

ColdBox Mode Listener

```
component{
    function configure(){}
    function beforeCacheFactoryShutdown(event, interceptData){
        // get factory reference
        var cacheFactory = arguments.interceptData.cacheFactory;
        // Do my stuff here:
        // I can use a log object because ColdBox is cool and injects one for me already.
        log.info@UDE, I am going down!@
    }
    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#@
    }
}
```

Standalone Mode Listener

```
component{
    function configure(cacheBox, properties){
        variables.cacheBox = arguments.cacheBox;
        variables.properties = arguments.properties;
        log = variables.cacheBox.getLogBox().getLogger()
    }
    function beforeCacheFactoryShutdown(interceptData){
        // Do your stuff here.
    }
    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#@
    }
}
```

Please note the *configure()* method in the standalone listener. This is necessary when you are using CacheBox listeners outside of a ColdBox application. The *configure()* method receives two parameters:

- **cacheBox**: An instance reference to the CacheBox factory where this listener will be registered with.
- **properties**: A structure of properties that passes through from the configuration file.

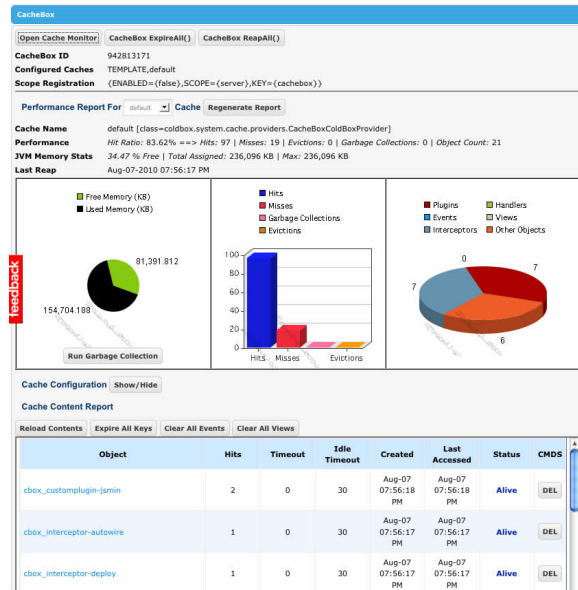
As you can see from the examples above, each Listener component can listen to multiple events. Now you might be asking yourself, in what order are these listeners executed in? Well, they are executed in the order they are declared in either the ColdBox configuration file as interceptors or the CacheBox configuration file as listeners.

Important: Order is EXTREMELY important for interceptors/listeners. So please make sure you order them in the declaration file.

Cache Reporting

CacheBox comes with great reporting tools whether you are using it within a ColdBox application or standalone. This section deals with how to do a-la-carte reporting by using the custom tags that CacheBox ships with. Below is the simplest form of usage for the monitor reporting tags:

```
<!-- Import Report Tags: use /cachebox/system if using standalone -->
<cfimport prefix="cachebox" taglib="/coldbox/system/cache/report" />
<!-- Create CacheBox with default configuration -->
<cfif structKeyExists(url,init) OR NOT structKeyExists(application,cacheBox)
<cfset application.cacheBox createObject(component="/coldbox/system/cache/CacheFactoryInit")
<cfelse>
<cfset cachebox = application.cacheBox
</cfif>
<cfoutput>
<html>
<head>
<title>CacheBox Monitor Tool</title>
</head>
<body>
<!-- Special ToolBar -->
<div id="toolbar"
<input type="button" value="Reinit" onclick="window.location='index.cfm?reinit'"
</div>
<!-- Render Report Here -->
<cachebox:monitor cacheFactory="#cacheBox#"
</body>
</html>
</cfoutput>
```



That's it! You basically import the tag library from `/coldbox/system/cache/report` or `/cachebox/system/cache/report` and then use the `monitor` tag to render out the monitor. What's cool about the monitor is that it is completely skinnable. Please see the *CacheBox Report Skins* for more information. Let's check out the attributes for this custom tag:

Attribute	Type	Required	Default	Description
<code>cacheFactory</code>	<code>coldbox.system.cache.CacheFactory</code>	true	---	The reference to the CacheBox factory to report on.
<code>baseURL</code>	string	false	<code>cgi.script_name</code>	The location of the script so the tag can create links for Ajax calls and rendering calls.
<code>skin</code>	string	false	<code>default</code>	The name of the skin to use for rendering the report. The skins are found at <code>/coldbox/system/cache/report/skins</code> .

Important: Each skin can implement its own attributes, so always check the skins documentation to see what extra attributes it implements.

Here are some examples of the tag usage:

```
<<cacheboxmonitor cacheFactory=#cacheBox#>
<<cacheboxmonitor cacheFactory=#cacheBox#baseURL=#index.cfm?event=cacheMonitor>
<<cacheboxmonitor cacheFactory=#cacheBox#skin=#coolkiy>
<-- Embedding report in a coldbox application's admin view ---->
<<cacheboxmonitor cacheFactory=#controller.getCacheBox()##*
baseURL=#event.buildLink(event.getCurrentEvent())#*
```

CacheBox Report Skins

News: We currently have a \$75 Amazon Gift Card contest! Just build a skin and the best one wins! <http://blog.coldbox.org/most-cfm/cachebox-monitor-skins-contest-win-75-amazon-gift-card/>

CacheBox also allows for the creation of reporting skins so you can create gorgeous looking reports for its caches, configurations, content, etc. The location of such skins is in the following location:

```
/coldbox/system/cache/report/skins
or
/cachebox/system/cache/report/skins
```

The name of the folder inside of the *skins* folder is the unique name used to specify the skin in the custom tag. You can look at the *default* skin to learn from it and see how you can build skins yourself.

Skin Attributes

A skin receives attributes or configuration elements via the custom tag used for the report monitor. Basically any attribute you add to the custom tag will be available in the skin's pages in a structure called **attributes**, makes sense huh?

Tag Caller

A skin also receives a reference to the **caller** scope via a variable called, drum roll please, **caller**. So you can also reference the caller variables via this scope.

Skin Templates

The first step of creating skin templates is to create its holding folder inside of the *skins* directory. So if we were starting a new skin called *goodness* then you would create a new folder in the following directory:

```
/coldbox/system/cache/report/skins/goodness
or
/cachebox/system/cache/report/skins/goodness
```

The following templates are the ones you will be skinning and placing in this folder. In all reality you could potentially just have one, *CachePanel.cfm*. However, since you can bring in AJAX content to refresh certain parts of the panel, you break out its reporting functionality into various templates. The CFC in charge of rendering your skin templates is the *ReportHandler.cfc* located in the same *report* package, so we recommend also reading its API for more in depth information. So let's explore them:

Template	Required	Description
<code>cachebox.js</code>	true	The JavaScript file that will be automatically loaded into the header content via a <code>cfhtmlhead</code> call. You can put any JavaScript you like here or load more JavaScript files via your skin templates.
<code>cachebox.css</code>	true	The css file that will be automatically loaded into the header content via a <code>cfhtmlhead</code> call.
<code>CachePanel.cfm</code>	true	The main template that displays the report monitor to the user. This skin could potentially hold action buttons and other parts of the cache report rendered in specific locations by using rendering methods (see <i>ReportHandler</i> section).
<code>CacheReport.cfm</code>	false	This template is usually rendered via the <code>renderCacheReport(cacheName)</code> method and it is supposed to render out a report of the cache provider using the incoming <code>cacheName</code> argument. This template usually has a call somewhere for the content report of such cache provider via the <code>renderCacheContentReport(cacheName)</code> method.
<code>CacheContentReport.cfm</code>	false	This template is usually rendered via the <code>renderCacheContentReport(cacheName)</code> method and it is supposed to render out a report of the contents of the cache provider using the incoming <code>cacheName</code> argument. This table of contents can also have action buttons assigned to them.

Note: All skins are rendered within the *ReportHandler* component. This means that you have access to this object's methods and local variables. We recommend you look at the default skin's templates for usage.

ReportHandler

This object is in charge of rendering skin templates and also executing processing commands. The custom tag creates this object and prepares it for usage, so don't worry about it, just know how to use it. The following are the variable compositions this object has and therefore you can use them in your skin templates:

Variable	Type	Description
<code>cacheBox</code>	<code>coldbox.system.cache.CacheFactory</code>	A reference to the running CacheBox cache factory
<code>baseURL</code>	string	The <code>baseURL</code> attribute passed via the tag configuration
<code>skin</code>	string	The <code>skin</code> attribute passed via the tag configuration
<code>skinPath</code>	string	The non-expanded path to the skin in use. e.g. <code>/coldbox/system/cache/report/skin/MyCoolSkin</code>
<code>attributes</code>	struct	A reference to the attributes structure passed via the tag
<code>caller</code>	template	A reference to the caller page of the custom tag.

This tag also has the following methods that you might be interested in:

Return Type	Method
void	<code>processCommands([command=],[cacheName=default],[cacheEntry=])</code> Execute and process a cacheBox command
any	<code>renderCachePanel()</code> Render the <i>CachePanel.cfm</i> template
any	<code>renderCacheReport(cacheName)</code> Render the <i>CacheReport.cfm</i> template which renders typically the report information about a specific cache provider

```
any      renderCacheContentReport(cacheName)
         Render the CacheContentReport.cfm template which typically renders the report of the content of the cache provider

any      renderCacheDumper(cacheName,cacheEntry)
         Tries to retrieve the cacheEntry from the cacheName provider and dumps it
```

For example, here is a snippet of the *CachePanel.cfm* template to designate where the cache report for a specific cache provider will be rendered by default:

```
<!-- Named Cache Report -->
<div id=#fw_cacheReport#><renderCacheReports/></div>

Since no cacheName argument is provided, it will use the default value of default. Here is a snippet of the cache report template of where it designates the content report to be rendered. It also verifies that the cache provider has reporting enabled and uses a custom attribute called contentReport.

<!-- Content Report -->
<ifcacheProvider.isReportingEnabled() AND attributes.contentReport
<#cacheContentReport>

<!-- Reload Contents -->
<input type="button" value="#Reload Contents"
name=#cachebutton_reloadContents#
style=#out-size:10px#
title=#load the contents#
onclick=#fw_cacheContentReport("#URLBase#", "#arguments.cacheName#" )#

<!-- Expire All Keys -->
<input type="button" value="#Expire All Keys"
name=#cachebutton_expirekeys#
style=#out-size:10px#
title=#expire all the keys in the cache#
onclick=#fw_cacheContentCommand("#URLBase#", 'expirecache', '#arguments.cacheName#" )#

<!-- ColdBox Application Commands -->
<ifcacheBox.inColdBoxLinked()
<!-- Clear All Events -->
<input type="button" value="#Clear All Events"
name=#cachebutton_clearallevents#
style=#out-size:10px#
title=#remove all the events in the cache#
onclick=#fw_cacheContentCommand("#URLBase#", 'clearallevents', '#arguments.cacheName#" )#

<!-- Clear All Views -->
<input type="button" value="#Clear All Views"
name=#cachebutton_clearallviews#
style=#out-size:10px#
title=#remove all the views in the cache#
onclick=#fw_cacheContentCommand("#URLBase#", 'clearallviews', '#arguments.cacheName#" )#
</if>

<!-- Loader -->
<span class=#fw_redText fw_debugContent#fw_cacheContentReport_loader# Wait, Processing...>
<div class=#fw_cacheContentReport#fw_cacheContentReport#
#renderCacheContentReport(arguments.cacheName)#
</div>
</if>
```

Action Commands

Each skin template can execute action commands. We have several already constructed into the tag that can execute if it detects an incoming URL variable called **URL.cbox_command**. So if your report exists in a page called *monitor.cfm* then just call that same page (via AJAX preferably) with some URL variables attached (See default skin example). Now, below are the ones we implement, but you can build as many as you like and place them in the page where you use the monitor tag or in the skin templates themselves, wherever they make sense to you.

The following are the commands built in to the reporting tag and the incoming URL variables it expects. Please note that the command is taken from the **URL.cbox_command** variable:

Command	URL Variables	Description
expireCache	<ul style="list-style-type: none"> url.cbox_cacheName 	Executes a expireAll() in the cache provider specified by the incoming cache name.
reapCache	<ul style="list-style-type: none"> url.cbox_cacheName 	Executes a reap() in the cache provider specified in the incoming cache name
delCacheEntry	<ul style="list-style-type: none"> url.cbox_cacheName url.cbox_cacheEntry 	Deletes the passed in cache entry from the named provider
clearAllEvents	<ul style="list-style-type: none"> url.cbox_cacheName 	Executes a clearAllEvents() in the cache provider specified in the incoming cache name(Must be a ColdBox enabled cache)
clearAllViews	<ul style="list-style-type: none"> url.cbox_cacheName 	Executes a clearAllViews() in the cache provider specified in the incoming cache name(Must be a ColdBox enabled cache)
cacheBoxReapAll	<i>none</i>	Executes a reapAll() via the CacheBox Cache Factory
cacheBoxExpireAll	<i>none</i>	Executes a expireAll() via the CacheBox Cache Factory
gc	<i>none</i>	Executes a suggestion to the JVM to produce a garbage collection

Note : If the tag detects an incoming command, it will execute it, reset the content and output a **true** to the response buffer.

Here are some sample calls:

```
GET monitor.cfm?cbox_command=gc
GET monitor.cfm?cbox_command=cacheBoxReapAll
GET monitor.cfm?cbox_command=delCacheEntry&cbox_cacheName=testKey
```

ColdBox Application Enhancements

As we have seen from the documentation, CacheBox can be used in both standalone mode and within ColdBox applications. If you are building ColdBox applications, which you should ;), then don't worry about creating CacheBox, it is already done for you. Just know how to configure it and how ColdBox leverages CacheBox for lots of internal key operations.

Please also refer to the [ColdBox Application Configuration](#) section to learn about the conventions used to discover how CacheBox should be configured in a ColdBox application.

ColdBox Caches

A typical ColdBox application is configured by default with two caches named: *default* and *template*. The *default* cache is used for data, objects, model objects, plugins, handlers, etc. The *template* cache is exclusively used for event caching and view fragment caching. This provides the developer a nice way to decide how each of these concerns are cached. So if you are in a multi-cluster environment and you wanted to share event and view fragment caching in the cluster, you could easily configure the *template* cache to use the *DDCStore* or the *DiskStore* and store these elements in a central location shared by the cluster. Lots of possibilities here.

Resources

We recommend seeing our other guides to see how caching is leveraged internally in ColdBoxEnabled

- [Event Handlers](#) : How event handlers are persisted and how event caching can accelerate your applications
- [Views/Views/Template](#) : A great way to find out how to do view fragment caching
- [Plugins](#) : How plugins are persisted if you are a plugin author
- [Interceptors](#) : Learn what are interceptors/listeners, how to use them and how to configure them
- [FlashRAM](#) : How the ColdBox Flash RAM can be configured to use CacheBox for distribution