

# CacheBox

---

*For ColdFusion*

By S. Isaac Dealey

[info@autlabs.com](mailto:info@autlabs.com)

## Contents

|   |    |
|---|----|
| Acknowledgements.....                             | 2  |
| What is CacheBox?.....                            | 3  |
| CacheBox Has PEPP.....                            | 3  |
| Resources.....                                    | 3  |
| Why CacheBox? .....                               | 3  |
| How Does It Work? .....                           | 4  |
| What Caching Engines Are Supported? .....         | 5  |
| How Easy Is CacheBox? .....                       | 6  |
| A Namespace Primer.....                           | 6  |
| Can I use CacheBox with Shared Hosting?.....      | 7  |
| What should I cache?.....                         | 7  |
| Getting Started.....                              | 8  |
| Installation .....                                | 8  |
| Hello Mr. Bond .....                              | 9  |
| Configuring the CacheBox Service (optional) ..... | 10 |
| What's In the Box? .....                          | 10 |
| Storage Contexts.....                             | 10 |
| CacheBox Agents.....                              | 11 |
| What Can I Do With An agent? .....                | 12 |
| Fetch and Store .....                             | 12 |
| Delete and Expire .....                           | 12 |
| How Do I Configure An Agent? .....                | 13 |
| CacheBox Nanny .....                              | 15 |
| The CacheBox Service .....                        | 16 |
| Monitor and Reap (Scheduled Task).....            | 16 |

|                                  |    |
|----------------------------------|----|
| Storage Types.....               | 16 |
| Custom Storage Types .....       | 16 |
| Configuration .....              | 17 |
| Eviction Policies.....           | 17 |
| The Management Application ..... | 18 |
| Home Page .....                  | 18 |
| Applications and Agents .....    | 19 |
| Cluster .....                    | 22 |
| Options.....                     | 23 |
| Advanced Topics .....            | 24 |
| Singletons.....                  | 24 |
| Session Cache.....               | 25 |

## Acknowledgements

Great software is never the product of a single individual. Many thanks go to [Rob Brooks-Bilson](#) who's excellent advanced caching techniques presentation for the Adobe Max conference provided invaluable insight into the development of this project. Many thanks also to [John Hirschi](#), [Shayne Sweeney](#) and Tyler Smalley, who've done an excellent job integrating memcached with ColdFusion and on who's work the CacheBox integration of memcached depends. Additional thanks to Luis Majano and the [ColdBox](#) team for the development of the ColdBox cache with its administration tools and hot-swappable eviction policies, and to Team Mach-II for development of the [Mach-II](#) framework's configurable, custom storage types. The ideas and implementation of both of these tools informed development of the CacheBox project. If you find CacheBox useful, make sure you thank these guys. If you're feeling generous, send them something from their wish-list, we're sure they'll appreciate it.

### Contributors and Beta Testers

- [Steve Bryant](#)
- [Mike Henke](#)

## What is CacheBox?

CacheBox is a hassle-free, advanced caching solution for any ColdFusion project. There are many other options for caching in ColdFusion. Each option has different syntax and a different set of features making them easy to apply but inconvenient to change. CacheBox is the only solution that provides all the features of the other caching options, plus a few of its own unique features. Of course you could try all the other caching options (that would take a long time and wouldn't be very rewarding) or you could install CacheBox and try them all within minutes, see everything in one place and switch them instantly with hot-swapping storage types. And thanks to its Agent / Service design, CacheBox is also portable, allowing you to create CacheBox-ready applications with no dependencies.

## CacheBox Has PEPP

**P**ortable – Agent / Service design means no dependencies – use it everywhere!

**E**xtensible – Create custom storage types and eviction policies to fit your needs.

**P**ersonal – Cache is configured at run-time, using your server's available resources.

**P**owerful – Cluster synchronization, management application, hot-swappable storage types, etc.

## Resources

For additional support or to contribute to the CacheBox project, join our discussion group at <http://groups.google.com/group/cfcachebox>.

## Why CacheBox?

Although many of us have written a lot of caching code over the years, most of us don't really enjoy it. Caching features aren't something we look forward to working on, it's something we have to get out of the way to work on other more interesting things. My inspiration for CacheBox came when our lead developer on a job in Boston expressed frustration with a new ColdBox project. He wanted to install the ColdBox framework and have it just work. Instead he seemed frustrated by a need to manually configure the framework's object caching. Getting it to behave the way he wanted didn't seem as easy as he'd hoped. I thought to myself, "why can't it just work?" Why should we have to keep reinventing the wheel? Caching is not a new concept, in fact, it's pretty well-worn territory. So why after all this time hasn't someone created a single pluggable caching system that could work for all these frameworks and applications and save us all the headache? That's the goal of CacheBox, to provide caching that "just works". If we've done our job right, it should be a hassle-free solution you can use on every project.

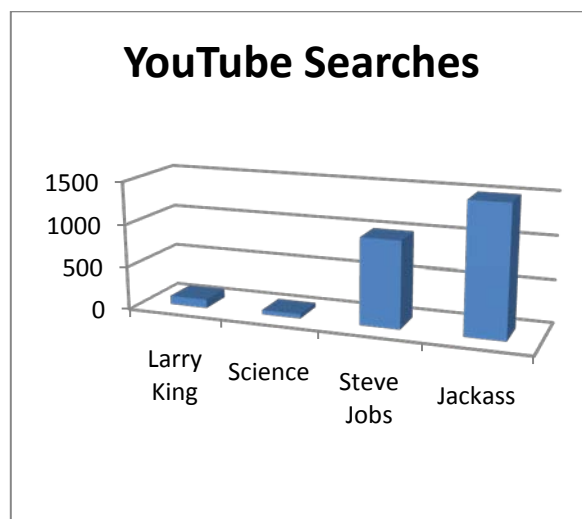
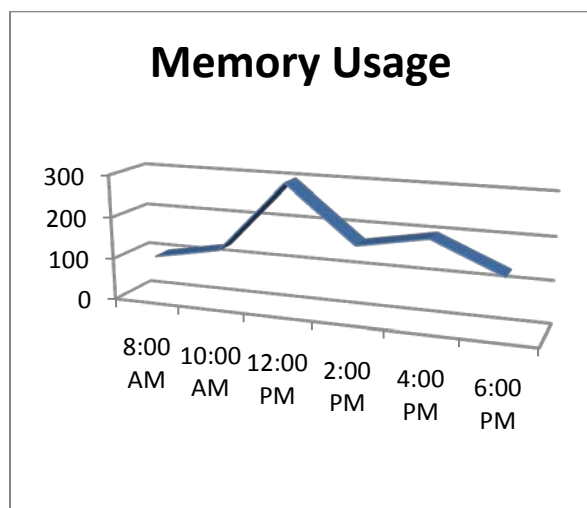
It's not my intention to criticize ColdBox here; in fact of all the frameworks for ColdFusion, ColdBox has possibly the most complete and eminently usable caching system. CacheBox expands on that, combining many of the ideas in the ColdBox cache with many of the other caching options for ColdFusion such as Memcached and the extensible caching features in Mach-II. It does all this and also makes it all portable for use with any framework or application. If you're looking at caching tools for ColdFusion, you probably want to improve performance of an application, or ensure it can handle some heavy traffic (or both). So you look into the different caching options that are available to you, both natively and third-party solutions like memcached. The question is which one do you choose? Each solution is designed for

a particular kind of caching problem and they all have different strengths and weaknesses... so which one do you choose for your project? Why not try them all? That's a lot of options, and testing each one can be a daunting and time-consuming task with little reward. Normally it would require rewriting a fair amount of code to test each one. How would you like a tool that lets you test and use any or all of them, without changing any of your code? CacheBox does this by hot-swapping different caching strategies at run-time and more...

## How Does It Work?

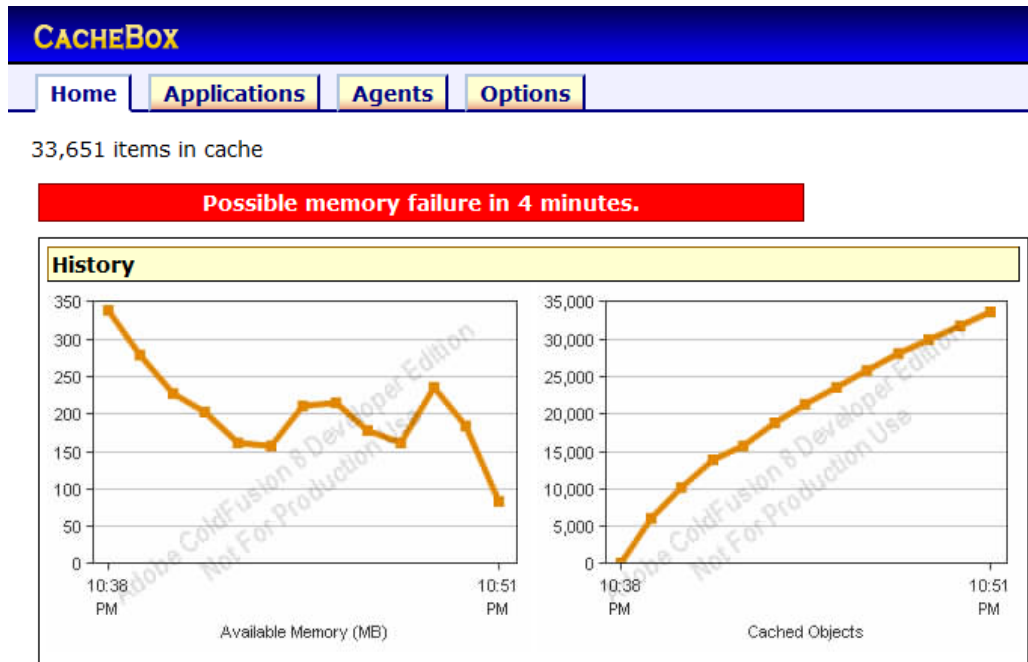
Work less and get more done. Are you skeptical? Let me ask you a few questions. How much do you know about your cache? Do you know what's in your cache right now? Do you know how long it's been there? Do you know how often people request it? Do you know what your server's resources look like throughout the course of a day? Do you know what a demand graph for your content would look like?

You probably answered no to a few of those questions, because the reality is that as a developer, you don't have time to monitor your server's resources and cache utilization all day. Even if you did, most of the available caching systems don't tell you much (if anything) about your cache, so although you might know that you have less than 100 ColdFusion queries cached, you have no clue which queries they are, how often they're requested, or how full your cached-query queue is. Because we don't really get to see into our caching systems and because our work time is a premium, we tend to build linear caching systems. For example, ColdFusion query caching is a linear system – it holds up to  $N$  queries in queue and when it reaches  $N+1$ , it removes the oldest one from the queue. But these linear strategies are inefficient because we're using them to address a non-linear problem. System resources like available memory and processor utilization are non-linear, and demand for content is also non-linear.

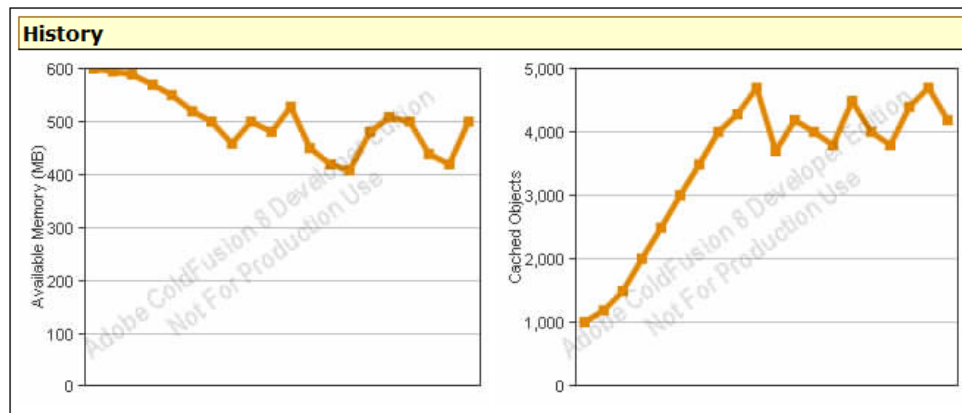


However although you can't constantly monitor your server's cache (even if you wanted to), there is someone who can. The server can monitor itself. By giving the server the intelligence to not only log but act on these non-linear events such as content popularity and traffic spikes, the server is able to automatically construct an optimal, non-linear solution to match the non-linear challenge of cache management.

So if the server detects a potential disaster like this:



It can analyze the problem, optimize the cache and avert the disaster on its own like this:



Because the server can monitor itself much more frequently and more consistently than you or I can, it can make incremental decisions about resources throughout the day to provide much more informed, accurate and efficient cache optimization.

### What Caching Engines Are Supported?

By default, CacheBox includes support for caching in memory, to file, to database, Java SoftReferences, Railo's Cluster scope, the native ehCache instance added in ColdFusion 9 and Memcached (requires the [cfmemcached](#) project also). Support for additional caching engines is also easy to add.

## How Easy Is CacheBox?

Are you wondering how many hours of studying you'll have to do to get CacheBox working? Don't worry. CacheBox is very easy to use. You will need to be able to create ColdFusion Components (CFC), and you need to have a basic understanding of namespaces and that's about it. Later on, you may want to learn more about in-process versus out-of-process caching because CacheBox supports both side-by-side however, you don't need to understand them right now.

### A Namespace Primer

A namespace is a fancy way of saying "a way to identify things". You can think of a namespace as the street you live on, where each house has its own number or "name". So if you live at 1313 Elm Street, then your house is the only house that can be at 1313. In order for another house to be at 1313, it would have to be on another street (or in another "namespace"). Just like street names and house numbers help the post office find your house to deliver your mail, namespaces and names help the computer store and find your content. And namespaces also tend to be nested. So for example, in the namespace of the world, there is only one country named "USA" and only one country named "Canada". Our country then has a namespace within it, so for example in the USA there is only one state named Texas and only one state named California. And each state has another namespace within it, so in Texas there is only one city named Dallas and only one city named Houston.

So my individual house might be at 1313 Elm Street, Dallas, TX, USA. Of all the places in the world, there is only one house at that address, which is made up of five individual namespaces.

House Namespace

- 1313

Street Namespace

- Elm Street

City Namespace

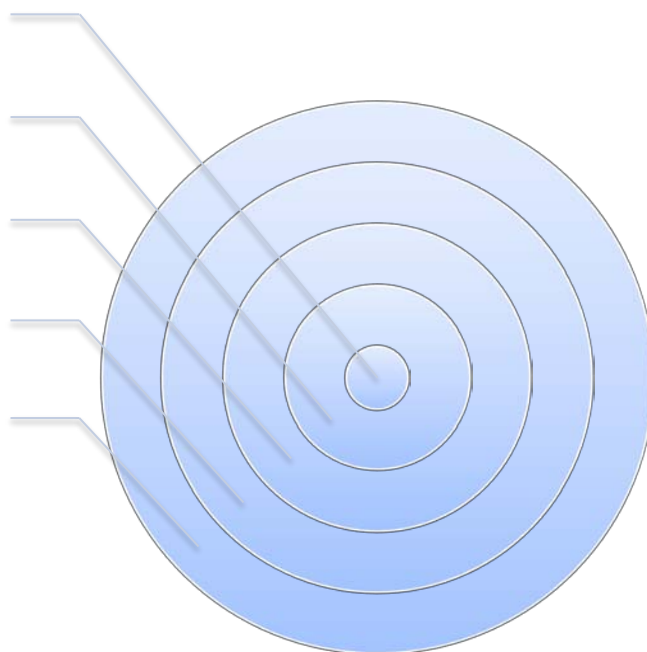
- Dallas

State Namespace

- Texas

Country Namespace

- USA



So any time you have a structure like any of the ColdFusion server's built-in scopes (server, application, session, arguments, etc.), you have a namespace.

What would it look like if you tried to put two houses at the same spot in the middle of your street? It would probably look like a mess and it wouldn't work very well. When you try to put two things at the

same space within a namespace, for example, two houses at 1313 on the same street, that's called a "namespace collision" or "namespace conflict". You may also hear this called a "key collision" because the names and the things they represent are often called "key / value pairs".

### Can I use CacheBox with Shared Hosting?

Yes. CacheBox is designed to allow multiple copies per server without namespace collisions or other problems. The management application will only show content for your copy of CacheBox and because the management data is separate, optimization may not be as efficient. This should not prevent you from running CacheBox in a shared environment.

### What should I cache?

That's a good question. It's also one that doesn't have a straightforward answer. My first caveat is this: you should manage anything you can through CacheBox. So if you're using query caching in ColdFusion now and you add CacheBox, the framework will work better once you switch all your cached queries to CacheBox instead of native query caching in ColdFusion. Although this does mean slightly more work at first when you convert your query caching, it will also mean better performance in the long-run. ColdFusion template caching on the other hand can't be managed through CacheBox, so this rule of thumb doesn't apply to template caching.

In his Adobe Max session on Advanced ColdFusion Caching Strategies in 2008, Rob Brooks-Bilson suggested a rule of thumb that you should cache as late as possible. So if you can cache an entire page, cache the entire page. If you need dynamic content in the middle of the page, then of course this isn't possible, but you should execute as much code as possible before caching the result. I think this advice might have been helpful to me several years ago when I developed a caching strategy for a complex ecommerce pricing system that turned out ultimately to be too slow to use. It's important to remember that the caching code itself will take time to execute and sheer speed isn't always its first goal. If I had been thinking about caching later, I might have developed a much better caching strategy for our ecommerce system.

On the other hand caching late often means caching content (usually html page fragments), meanwhile there are other cases in which caching database queries or objects (as in the case of the Transfer ORM) are also quite useful.

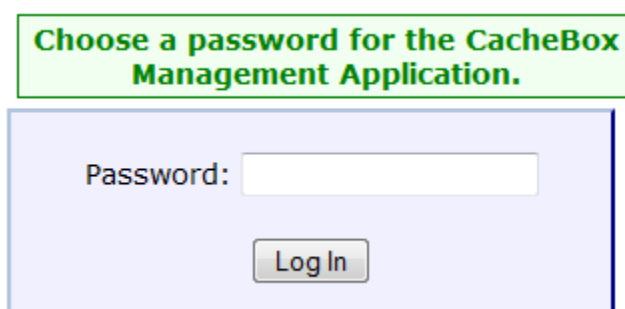
Caching is most effective for content that is viewed often and changes infrequently. Content that is changed too frequently needs to be updated in cache and may cause more problems than simply not caching it. Content that is viewed too infrequently will simply waste memory or other resources in cache.

## Getting Started

This section describes the essential requirements to get started with CacheBox. After reading this section you should know how to install CacheBox, some optional configuration options, and how to store and retrieve data from the cache.

### Installation

I'm something of a stickler when it comes to making things easy to use. That's why I've always insisted that the installation instructions for the onTap framework would have one and only one step: "unzip it". The same is true for CacheBox, with one minor exception. The CacheBox framework includes a management application that allows you to examine the content of your cache at run-time. While this transparency is good for you as a developer, it could also be a security risk, which is why it includes a password. After you've extracted your copy of CacheBox, you will need to view the management application in a browser and set the password. You should always make sure you set the password immediately after placing your CacheBox installation because anyone could set it and get into your cache until you do. If you need to reset the password, simply delete the file `/settings/password.cfm`.



The image shows a web interface for the CacheBox Management Application. At the top, a green-bordered box contains the text "Choose a password for the CacheBox Management Application." in green. Below this, a light blue box contains a "Password:" label next to a white text input field. At the bottom of the blue box is a "Log In" button.

In most cases your CacheBox management application is accessible at <http://127.0.0.1/cachebox/>. If your installation is not accessible at this URL, you will also need to configure the `pollingURL` in `config.cfc`. If you're like most people, you've just tried to open `/cachebox/config.cfc` and discovered that you can't because the file doesn't exist. Don't panic. ☺ Read "Configuring the CacheBox Service" following the Hello Mr Bond example on the next page.



## Hello Mr. Bond

The only thing you need to start using the CacheBox is an agent (/cachebox/cacheboxagent.cfc). You will need to give your agent a unique name to make sure that the CacheBox service doesn't confuse your agent with other agents on the server. Once you've done that, then you can fetch and store content through your agent, using a unique identifier for each content item you want to store. Items are placed in cache using the *STORE* method and retrieved from cache using the *FETCH* method. This hello world example shows how to use an agent named "James\_Bond" to store intelligence using the code key "secret\_x".

```
<!--- Call our agent, James Bond --->
<cfset agent = CreateObject("component",
    "/cachebox/cacheboxagent").init("James_Bond") />

<!--- Tell James the secret --->
<cfset agent.STORE("secret_x", "Hello World") />

<!--- Villains capture Bond and force him to talk! --->
<cfset message = agent.FETCH("secret_x") />

<!--- Make sure the message is legitimate --->
<cfif message.status eq 0>
    <!--- The message was retrieved okay! --->
    <cfoutput>#message.content#</cfoutput>
<cfelse>
    <!--- The message was not found in cache --->
    ... Kill Mr. Bond! ...
</cfif>
```

Once you've executed this sample code from an application on your server, you can then view the CacheBox management application at <http://localhost/cachebox> (or wherever you placed your copy of the CacheBox framework). Within the management application you can view all the agents that have registered with the CacheBox service, analyze their content, remove content from the cache (one at a time or for an entire agent or application), and configure various storage types and eviction policies for your agents.

Use this information wisely; the fate of the free world is in your hands!

## Configuring the CacheBox Service (optional)

In most cases you shouldn't need to configure the CacheBox framework, it should just work. If you have an unusual use case in which you do need to tweak or modify the CacheBox framework, it should be easy to do that also.

If you're like most people, you might look at the `Application.cfc` and see that it creates a `cacheboxservice.cfc` and pulls a configuration object from there. The configuration object is `defaultconfig.cfc`. You might be tempted to edit the `defaultconfig.cfc` file; **DON'T**. Like the `onTap` framework, CacheBox is designed with future-proofing in mind. Any changes you might need to make to the framework should be possible via the `config.cfc`. All you have to do is create the file as a CFC that extends `defaultconfig.cfc` and anything you need to change about the framework can be overridden in `config.cfc`. This will protect you from most changes in future versions of the CacheBox core framework.

Your CFC should look like this:

```
<!-- /cachebox/config.cfc -->
<cfcomponent extends="defaultconfig">
    <!-- your custom code here -->
</cfcomponent>
```

Information about some of the configuration tweaks you might like to make can be found in the comments in `defaultconfig.cfc`.

## What's In the Box?

The Hello Mr. Bond example in the previous section shows you just a small glimpse into the CacheBox. There you see how cache is stored and retrieved from the service through an agent, and you get to see the management application. What you haven't seen is the service object (`cacheboxservice.cfc`) that manages everything, configurable storage types and eviction policies and how all these things work together. This section gives you an in-depth look at what's in CacheBox and what you can do with it.

## Storage Contexts

When you create a CacheBox Agent, you give it a name and a context (default is *APPLICATION*). Both of these properties identify the namespace within which the agent stores content, the same way your street address identifies your house. Supported contexts include *CLUSTER*, *SERVER* and *APPLICATION*. Agents in separate applications will share the same namespace and content if they have the same name and are on the same server (in the *SERVER* context) or in the same cluster (in the *CLUSTER* context). If you change the name or context of an agent during development, it effectively becomes a different agent and loses all of its configured settings and content.

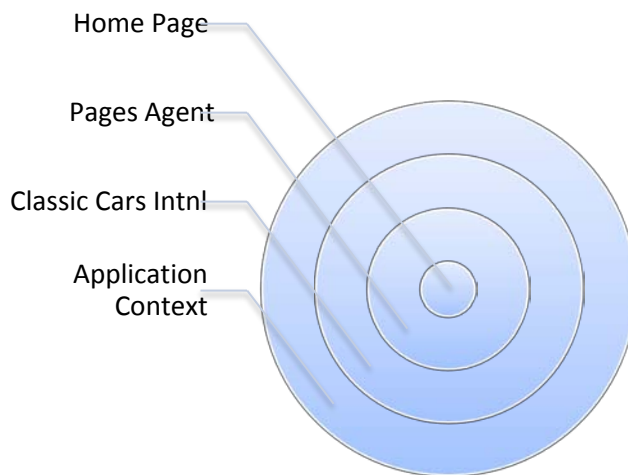
So for example if you create an agent in the *APPLICATION* context named “Pages” and then store some HTML with that agent using the name “Home\_Page”, you then have your home page content stored within the *APPLICATION* -> Pages namespace.

```
<!-- Create an APPLICATION Agent -->
<cfset agent = CreateObject("component",
    "/cachebox/cacheboxagent").init(
    AgentName="Pages",
    Context="application") />

<!-- Store the Home Page with our agent -->
<cfset agent.STORE("Home_Page",HTML) />
```

What this means is that content stored in the “Pages” agent is unique to the application where it was created. So if you have two clients, Classic Cars International (CCI) and Ferrari World (FW), both sites can be hosted on the same server and each can have the same “Pages” agent and different content stored for the “Home\_Page” cache entry, because the content is unique to each application.

If you want to share content between these two applications, you can easily do that by creating the



agent in the *SERVER* context instead. For example, you might want to cache your hosting company’s “server\_status” page in another “Pages” agent in the *SERVER* context and both sites would share the same cache for that page. If you want the same content to be shared between multiple servers, simply create the agent in the *CLUSTER* context. Each server in the cluster must have its own copy of CacheBox installed.

Although the CacheBox service provides many methods of storing cache, not all storage types can support all contexts. For

example, the *CLUSTER* context allows multiple servers to share cached content between them, which requires an external storage method (such as a memcached server), which may not be available. When a requested context is not available, CacheBox gracefully degrades down to the next available context. So for example, your agent may request to store cache for the *CLUSTER* context, but if there is no available storage for that context, the service will use the *SERVER* context instead. If there is no CacheBox service at all, the agent will degrade down to the *APPLICATION* context and manage its own cache.

## CacheBox Agents

In an ideal environment, the only thing your application knows about CacheBox is its agents (cachebox/cacheboxagent.cfc). Each application may have many agents for different kinds of content, and they will all connect to a single service object on the server. So for example full page content may be stored with a “Pages” agent, while an ORM like Transfer might cache data access objects (DAO) in a “Transfer” agent. The agents do two things, first they simplify your application code and secondly they

restrict your application's control of the cache. Both of these things mean less work for you. The reason the agent restricts control of the cache is to limit your application's dependency on the CacheBox framework and to channel most cache management through the framework and management application. When the CacheBox service is installed, the agent behaves like an advisor, offering suggestions about how cache should be handled, rather than imposing rules or managing the cache directly. When the service is not installed or not available, the agent manages its own cache while ignoring contexts and eviction policies.

Agents for your application should be created once and placed in the application scope, instead of creating them on each request. Although this should improve performance, the primary reason for keeping agents in the application scope is to allow them to manage cache if they are unable to connect to the CacheBox service because it is not installed or for some other reason.

### What Can I Do With An agent?

- Add content to cache: `agent.store(cachename,content)`
- Retrieve content from cache: `agent.fetch(cachename)`
- Remove an individual piece of content from cache: `agent.delete(cachename)`
- Remove several related pieces of content from cache: `agent.expire(cachename & "%")`
- Remove all the agent's content from cache: `agent.reset()` or `agent.expire()`
- Check if the agent is connected to a CacheBox service: `agent.isConnected()`
- Miscellaneous: check the agent version number, access the service object, check the applied context and applied eviction policy (which may be different from the requested context or eviction policy), etc.

### Fetch and Store

Most of the work you do with CacheBox will be done through the *STORE* and *FETCH* methods, which either add content to the cache or return content from the cache respectively. Just as you need to provide a name for your agent, you also need to provide a name for your content. Other caching systems usually refer to this as a "key", CacheBox refers to it as a "cache name". The name is required to uniquely identify your content in the cache agent's namespace, otherwise you won't be able to get your content back out, just like the post office won't be able to deliver mail to your house unless the address includes the house number on your street. The cache name only needs to be unique within the agent, so if your agent is named "pages", it's okay for the cache name to be a `page_id` variable, it doesn't need to be "page\_#page\_id#".

Both the *STORE* and *FETCH* methods return a structure with two keys, a status key and a content key. The status key is a numeric value that may indicate a potential failure that might have occurred. A status of zero (0) indicates that the fetch or store operation worked perfectly. A status of one (1) indicates that the content was not found in storage. Mostly you need to know that a status of anything other than 0 returned from a *FETCH* operation indicates that the content is not yet cached and needs to be generated. If you are storing singleton objects, skip to the Advanced Topics section at the end of the documentation. If you don't know what a singleton is then you're probably not storing singletons.

### Delete and Expire

The agent component provides two methods of removing content from cache. These are with the methods *DELETE* and *EXPIRE*. In most cases it shouldn't matter which you choose. Both methods allow

you to remove multiple content items by using the percent symbol (%) as a wildcard character. So to remove all content with a cache name that begins with “product\_5\_”, you could use `agent.delete(“product_5_%”)` or `agent.expire(“product_5_%”)`. In both cases, a *FETCH* operation immediately following this will not find any content for “product\_5”, so they’re functionally the same. The difference between these methods internally is that the *DELETE* method removes the content immediately, while the *EXPIRE* method only marks the content for later deletion. The importance of this is that while the *EXPIRE* method is likely to execute faster than the *DELETE* method, it doesn’t immediately free up the server’s memory. Instead the content stays in memory until the next reap-cycle. By default a reap cycle occurs every minute, so content won’t stay in memory for very long. So if you receive complaints from users about a page being slow and that page is deleting a lot of cache content, switching to the *EXPIRE* method may resolve that issue.

### How Do I Configure An Agent?

A CacheBox agent is configured through its *INIT* method, which provides several arguments for configuration. The only argument required when you create your agent is the *AgentName*, which uniquely identifies your agent and eliminates confusion with other agents, so it’s important to give your agents descriptive names. Below is the full list of init arguments for an agent.

**AgentName (Required):** Uniquely identifies your agent.

**Context:** The agent’s context describes the desired context for the cached content. Supported values are *CLUSTER*, *SERVER* and *APPLICATION*. When creating an agent, you should configure it with the largest possible context you think that agent might need in the future. For example, if you are a hosting company and are caching content pages for client sites, then the *APPLICATION* context is appropriate because each client will have their own content pages. On the other hand if you’re caching the display of videos that have been uploaded by users (like YouTube for example), then you should use the *CLUSTER* context because the application may eventually outgrow a single server and it’s better to be prepared by overestimating the need.

Unlike storage types and eviction policies (which are both hot-swappable), the context of your agents can NOT be changed after the agent is created. As previously mentioned, the service will gracefully degrade if you request a context that isn’t available, so the applied context of your agent may not be the same as the context you request. You can get the requested context of an agent with the *getContext* method and you can compare this to the context in use with the *getAppliedContext* method.

**Evict:** This argument allows you to declare an eviction policy for your agent. Eviction policies are ignored if the CacheBox service is not available. The default eviction policy is *AUTO*, meaning that the CacheBox service is free to assign an eviction policy it decides is optimal. Eviction policies of *FRESH* or *PERF* are auto-optimizing policies, with an emphasis on freshness of content or speed of delivery respectively. If you have an agent that needs permanent retention of its content, specify an eviction policy of *NONE*.

The remaining eviction policies include either a time limit or a cache-size limit. For example the *FIFO* policy means “first in first out” like the ColdFusion cached queries queue. To have a *FIFO* eviction policy, you have to tell the agent how large the queue is allowed to grow before it starts evicting content. If an eviction policy has a time limit or a cache-size limit, indicate the limit by appending a colon and number to the name of the eviction policy. For example, *FIFO:100* will store up to one-hundred items for the

agent, while AGE:20 will store the content for up to twenty minutes. You can get the requested eviction policy of an agent with the *getEvictPolicy*. If the agent is not connected to a CacheBox service, the *getAppliedEvictPolicy* method will return a value of *NONE* indicating that the eviction policy is being ignored due to lack of service.

**ReapListener:** This argument allows you to declare a component to receive notification when items are removed from cache. This object must contain one method named “ReapCache”, having two arguments named “CacheName” and “Content”. Using a reap listener adds dependency to the system and as such should be avoided if possible. If it is not possible to avoid using a reap listener, this example shows how to write your listener:

```
<cfcomponent displayname="ReapListener" output="false">
  <cffunction name="ReapCache" access="public" output="false">
    <cfargument name="CacheName" type="string" required="true" />
    <cfargument name="Content" type="any" required="true" />
    <!--- Perform any necessary clean-up with the content --->
  </cffunction>
</cfcomponent>
```

**CacheService:** This argument is provided primarily for sticklers who don't like the way CacheBox eliminates dependencies. It allows you to declare the CacheBox service to use when you create the agent and may also be useful for testing purposes.

**ApplicationName:** This argument indicates the name of the application to which this agent belongs. This is primarily for agents in the application context. In general, leave this argument alone. If you have some specific reason to need to fetch cache content from an alternate application, it might be useful to declare an agent into the other application, but it's not recommended.

## CacheBox Nanny

The CacheBox service is very efficient at handling groups of cache collected for an agent. The trade for this design is that it's not as good at handling custom eviction policies for individual content items. So for example, your "Pages" agent may have an eviction policy of age:20 or age:30, but you can't have an eviction policy of age:60 for just the "HomePage" item and age:20 for all the other pages. The nanny object is designed to give you more control over content expiration at the client side by wrapping itself around an agent and providing additional content expiration features that mirror the *CachedWithin* and *CachedAfter* attributes in the ColdFusion CFQUERY tag.

Here is an example of a nanny in use.

```
<!--- Create an APPLICATION Agent --->
<cfset agent = CreateObject("component"
    ,"cacheboxagent").init("testagent") />

<!--- Create a Nanny to enhance the agent --->
<cfset agent = CreateObject("component"
    ,"cacheboxnanny").init(agent) />

<!--- Store a message for up to ten seconds --->
<cfset span = CreateTimeSpan(0,0,0,10) />
<cfset temp = agent.store("hw","Hello World",span) />

<!--- lets monitor the storage --->
<cfloop condition="temp.status eq 0">
    <!--- wait for three seconds --->
    <cfset sleep(3000) />

    <!---
        Check the content and display it
        We should see "Hello World" 3 times
        The 4th dump should display empty content
    --->
    <cfset temp = agent.fetch("hw") />
    <cfdump var="#temp#" />
    <cfflush />
</cfloop>
```

## The CacheBox Service

CacheBox is designed somewhat like cable TV. When you rent a new apartment, it is usually “cable ready” when you move in, but the cable TV service is not available. You can of course still watch TV, you just don’t get access to all the cool cable stations like Discovery and the SciFi Channel unless you subscribe to the cable service. A CacheBox agent functions like a television set, giving you the ability to perform simple content caching. Once you install the CacheBox service then the agents stop managing their own cache and hand it off instead to the service object (`cachebox/cacheboxservice.cfc`), giving you access to all its advanced caching features. The service creates the config object, which in turn manages agents, storage types and eviction policies. Because it’s impossible to know which application will be accessed first when a server is started, the agents will start the service if it hasn’t been started already. This gives you the ability to eliminate the CacheBox framework as a dependency in your application. All you need to do is copy the `cacheboxagent.cfc` into your application and create objects of type “**`myapp.cacheboxagent`**” instead of creating objects of type “`cachebox.cacheboxagent`”.

## Monitor and Reap (Scheduled Task)

The primary concern of the service is to manage and optimize the cache, to allow for optimal performance of your applications. This means keeping frequently used content in cache to speed up delivery of pages, but it also means removing stale content from the cache to preserve the server’s available memory and prevent out-of-memory errors. By default stale cache is reaped to remove stale content using a scheduled task that’s updated by the `config.cfc` when the framework starts. The same scheduled task also updates the history statistics the framework uses to optimize the cache and that are displayed in the management application. Using a scheduled task for optimization and content expiration means that your site visitors are never waiting for long-running cache-management tasks to complete before they receive their content (and your very limited CFTHREAD pool is never being used for cache management either). If for some reason you don’t want to use the standard CFSCHEDULE method, you can override the `updateMonitoringTask` method in your `config.cfc`.

## Storage Types

The CacheBox service comes with a variety of storage types, some of which require configuration (FILE, DATABASE and MEMCACHED). Others work with no configuration (DEFAULT, SOFT, CLUSTER, NONE). Setting an agent’s storage type to NONE will disable caching for that agent (statistics will still be reflected in the management application, but content will not be stored).

## Custom Storage Types

You can expand the CacheBox framework to use a new type of storage by creating a CFC in the `cachebox/storage/` directory that extends the default storage type (`cachebox/storage/default.cfc`). The default storage type caches content directly in memory. An instance variable in the storage type CFC called `instance.context` indicates what contexts it supports (1=CLUSTER, 2=SERVER, 3=APPLICATION). All subsequent contexts are assumed to be supported also, for example, a storage type that supports the SERVER context is assumed to also support the APPLICATION context. The public variable `this.description` provides the description of the storage type that appears in the management application.



## Configuration

If a storage type needs configuration, you can declare the form to accept user input of configuration values in the component's *getConfigForm* method. The value returned should be a string of XML with a form in it, although it's not necessary to declare the form action, method, etc. and the input elements will be additionally formatted. Check the *FILE* storage type (`cachebox/storage/file.cfc`) for an example of how to collect configuration input. The name of the configuration file must also be declared in the *instance.settings* variable. Once the *getConfigForm* method is written and the *instance.settings* variable is set, the framework will handle the rest and you can configure the storage type in the management application. If you're familiar with XML, you might be tempted to configure storage types manually by editing the XML files; **DON'T**. Storage types are loaded once when the service starts and manually edited XML files will not be reloaded. The management application will update and reload the configuration.

## Eviction Policies

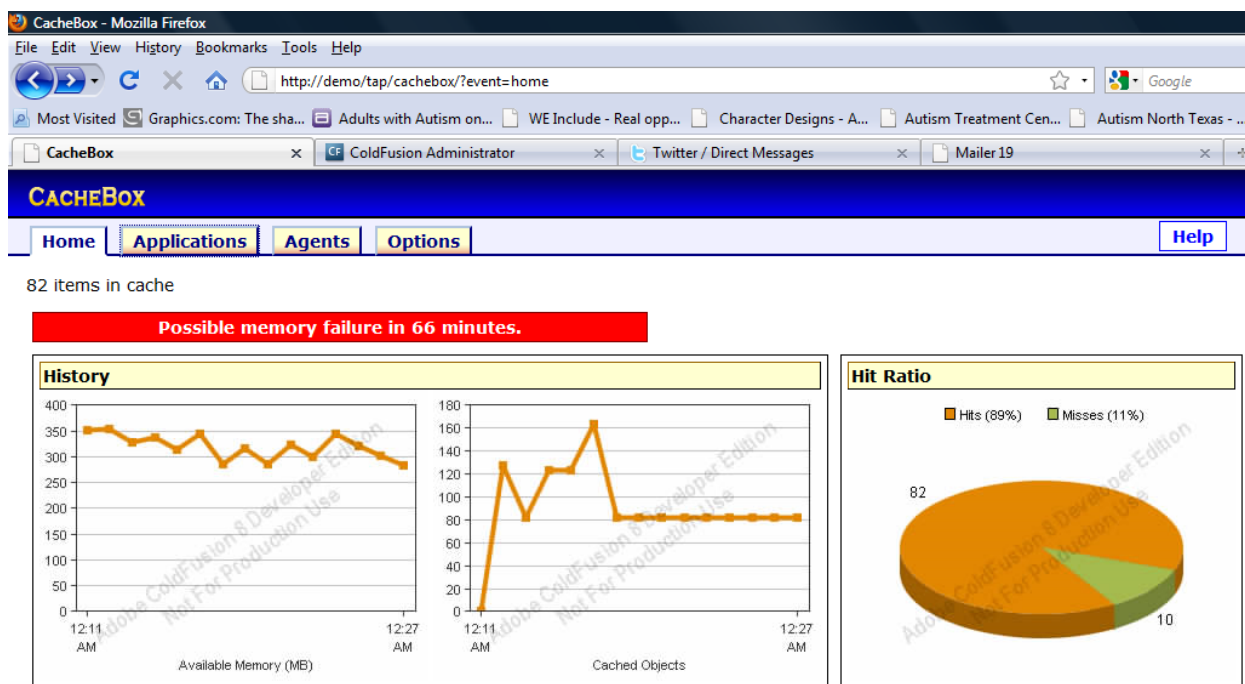
In addition to storage types, the CacheBox service comes with a variety of eviction policies for determining when content is stale and should be removed from cache. Like storage types, the CacheBox service can be extended with custom eviction policies by creating an eviction policy CFC in the `cachebox/eviction/` directory that extends `abstractpolicy.cfc`. The only thing this CFC needs to contain is the *getExpiredContent* method, which returns an array of index values from the provided cache query, to inform the `config.cfc` which items to remove. The cache argument passed to this function is a query containing only the non-expired cache content for the relevant agent. As with the storage types, the public *this.description* variable provides the description of the eviction policy in the management application. If the storage type requires a time limit or cache size limit, this information should be included in the description of the policy with the letter N representing the number of minutes or number of items in cache. For example, the idle eviction policy has a description of "Expires after N minutes unused". The placement of the N in the description tells the management application where to place an input element when configuring the storage type. The description must have white-space before and after the N. Eviction policies have no other configuration options in the current version of the CacheBox service.

## The Management Application

The purpose of the CacheBox management application is to give you better insight into and control over your cache. Yes, in an ideal world the CacheBox service would anticipate and handle any and all caching needs. In the real world sometimes when you build a better mousetrap, someone else builds a better mouse. So although we hope that the service will handle most applications with little or no configuration, the management application provides the extra tools you might need if you happen to have unusual requirements that call for manual configuration. Most of the management application should be fairly self-explanatory.

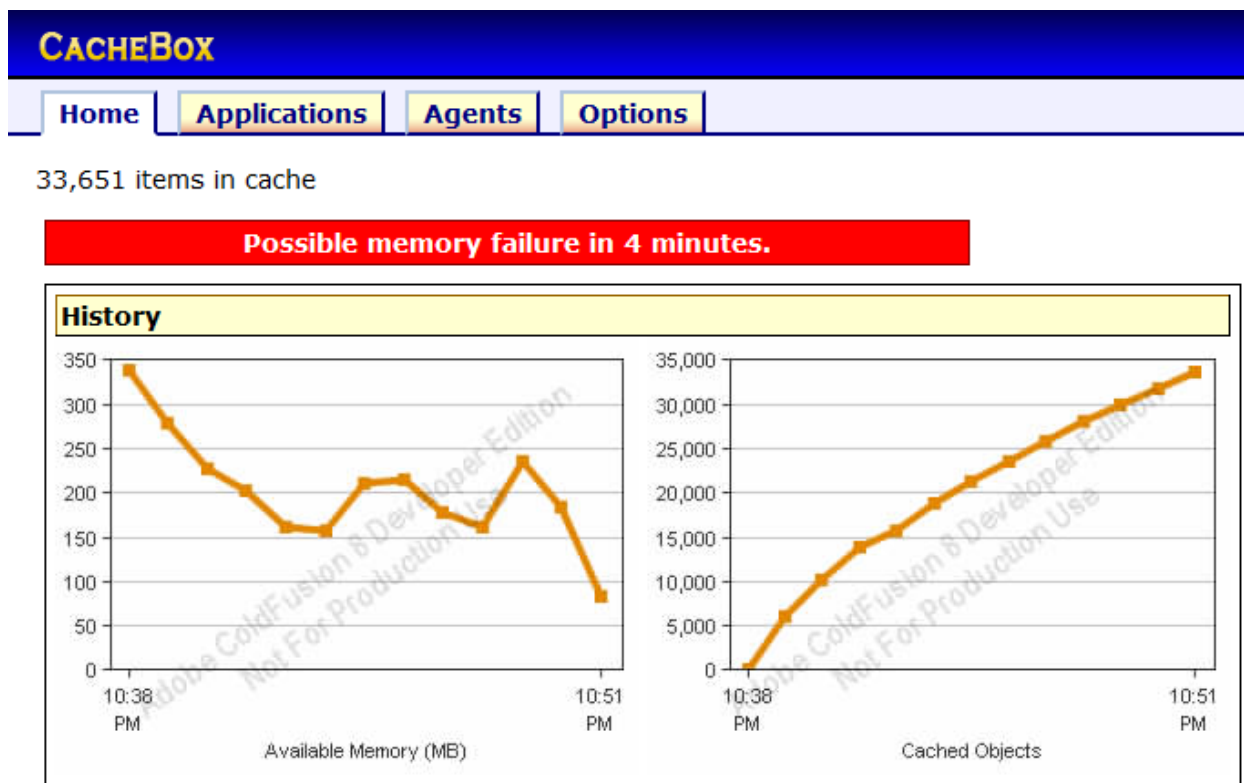
### Home Page

Upon opening the application home page, you'll see tabs at the top, a help link in the upper-right corner (which currently opens this PDF), a history graph showing available memory and cache usage for the past 20 minutes and a pie-graph showing the hit-ratio.



The charts on this page give you a very brief snapshot of current cache performance. You want the hit ratio to be high, but not necessarily 99%. It's okay for the hit ratio to be 70% or even lower if the service is evicting stale content, because evicting the stale content helps to preserve available memory. Because the service has access to all this data each time the service is polled (via a scheduled task), the service will know about potential problems before you do. It begins optimizing the cache agents and assigning new eviction policies once it predicts a possible memory failure within 2 hours. If after optimization the projection continues to drop to below 1 hour, the red warning message will appear on the CacheBox home page as you see above. (You can modify these times in config.cfc.) This warning may or may not be something you need to jump to address. In the example above, the service doesn't have quite a full 20 minutes of data for the prediction because it's only been running for 15 minutes. Less data means a less accurate prediction, however, even with a relatively accurate prediction, garbage collection and fluctuations in demand for the content can change the outcome. The warning is designed to make you pay attention, not necessarily leap to action.

If you do have a truly imminent memory failure, the obvious sign will be that the lines of the history graphs will point downward in the middle, as if they were creating a down-arrow to indicate “your system is going down”. Here’s an example of what that looks like:




If you open the management application and you see something like this, you should definitely start doing something about it right away.

### Applications and Agents

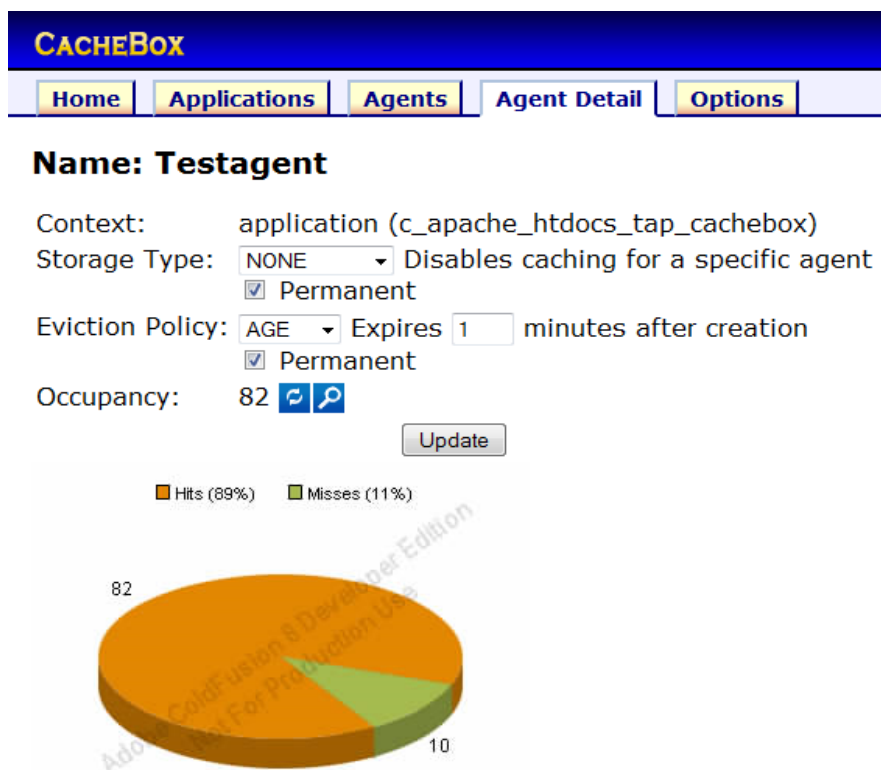
These tabs allow you to see lists of cache statistics for the agents that have registered with the service or the applications that have content stored on their behalf.



### Application

| Agent   | Occupancy | Oldest  | Last Hit  | Storage | Eviction |
|---|-----------|---------|-----------|---------|----------|
| c_apache_htdocs_tap_cachebox  |           |         |           |         |          |
|  testagent | 82        | 1:01 AM | 0 minutes | none    | age (1)  |

The blue row in this table shows the name of the application to which the “testagent” agent is assigned by its context. The button with the arrows allows you to flush the agent, removing all items stored on its behalf from the cache. The button with the magnifying glass allows you to view the details for the agent and optionally manually configure it.



This form shows stats and configuration for the agent you selected and allows you to manually configure it. Only storage types that support the agent's applied context will appear in this form. If you select the permanent checkbox below either the storage type or the eviction policy, that property will be stored in an XML file (settings/agents.xml.cfm) and used as the default value if the server is restarted. If you're familiar with XML, you might be tempted to manually edit this XML file; **DON'T**. Use this form, it will update the XML file and reload it into the application for you. To the right of the occupancy statistic, you see two blue buttons. The first button allows you to flush the cache for this agent. The second button with the magnifying glass allows you to view storage for this agent as you see below.













**CACHEBOX**

Home Applications Agents Options

**Content Of Agent: Testagent (C\_apache\_htdocs\_tap\_c**

Pages: [1](#) [2](#) [3](#) [4](#) [5](#)

Frequency: [Constant](#) [High](#) [Low](#) [Waste](#)

|   | Name   | Stored    | Last Hit  | Hits | Misses | Frequency             |
|---|--|-----------|-----------|------|--------|-----------------------|
| 1 |   content10      |           | 0 minutes | 0    | 1      |                       |
| 2 |   content10      |           | 0 minutes | 0    | 1      |                       |
| 3 |   content1054201 | 0 minutes | 0 minutes | 1    | 0      | <a href="#">1/min</a> |
| 4 |   content12      |           | 0 minutes | 0    | 1      |                       |
| 5 |   content13      |           | 0 minutes | 0    | 1      |                       |
| 6 |   content1322482 | 0 minutes | 0 minutes | 1    | 0      | <a href="#">1/min</a> |

Items in the list with zero hits are miss-counters for content that hasn't been stored. This may happen occasionally even when the application is working correctly, due to malformed URLs provided by site

visitors. To the left of each item in the list you see two blue buttons. The first button with the downward pointing arrow allows you to drop the cache for the specific record. The second button with the magnifying glass allows you to view the content for that record. If the content is a query, structure, array, object, binary or XML, this will display a dump of the object. If the content is text, it will display a form that allows you to modify the cache at run-time. If you are modifying cache at run-time, chances are there is another problem you should address in your application, but this will allow you to fix the symptom while you work on the problem.

**CACHEBOX**

[Home](#) | [Applications](#) | [Agents](#) | [Options](#)

### View Content

Agent: testagent (c\_apache\_htdocs\_tap\_cachebox)  
Content: content11

```

<cfcomponent output="false" extends="defaultconfig" hint="I provide the default settings for a Ca
<!-- the url to use for the cachebox scheduled task which monitors and reconfigures the
<cfset instance.pollingURL = "http://demo/tap/cachebox/monitor.cfm" />

<cffunction name="applyRequestSecurity" access="private" output="false">
    <cfargument name="targetPage" type="string" required="true" />
    <!--
        this makes the application inaccessible from a remote machine by default
        -- override this method in config.cfc to enable remote access and apply
        any custom security settings
    --->
    <!--
        *****
        *** not using 127.0.0.1 because of Apache issue with localhost ***
        *****

    <cfif cgi.REMOTE_ADDR is not "127.0.0.1">

```

## Cluster

The cluster tab provides a way to keep the CacheBox agents and storage types in your cluster in-sync. At the top of the page is a list of the currently declared servers in your cluster, showing their status and an ID for the server. The top-most item in the list is the server you're viewing currently (usually labeled "localhost"), highlighted to remind you that you're currently viewing it. Selecting the link on the name of a server will take you to the management application for that server (note that you will not be allowed to view the management application unless you have created a config.cfc and edited the *applyRequestSecurity* method). If there are no additional servers in the list, you can add them by entering the IP address or full URL to the CacheBox management application in the form below. Once entered, the servers will initially mistrust each-other and display a form to allow you to establish trust. Select the link to the alternate server, copy the ServerID from the top, press the back-button on your browser, paste the ServerID into the form and press the "Get Trust" button. Do this for each server in your cluster. If there are any servers left out, the system may not sync configuration across all the servers when you update a storage type or an agent.

**CACHEBOX**

[Home](#) | [Applications](#) | [Agents](#) | [Cluster](#) | [Options](#)

| Server                                     | Status      | ServerID                                   |
|--|-------------|--|
| <b>demo</b>                                | <b>OK</b>   | <b>C0252791-DFEE-3B0C-A6FC43EFE929A026</b> |
| <a href="#">192.168.1.101/tap/cachebox</a> | UNAVAILABLE |  |
| <a href="#">192.168.1.100/tap/cachebox</a> | OK          | C0252791-DFEE-3B0C-A6FC43EFE929A026        |

Enter one server per line as an IP Address or full URL.

```
192.168.1.101/tap/cachebox
192.168.1.100/tap/cachebox
```

## Options

The last tab provides two links to flush cache for the server or cluster contexts, as well as lists of the installed storage types and eviction policies. Storage types that require configuration will appear with a blue gear icon to the left of the type name. Selecting the linked name of the storage type will display the form to configure that type – each type will have a different configuration form. Not all of the installed storage types may be available. The ready column will indicate which storage types are available. Unavailable storage types will not appear in the agent configuration form, however, you may be able to make the storage type available if it requires configuration that has not been provided. The FILE storage type for example requires that you provide a directory name in which to store content before it is ready. The CLUSTER storage type on the other hand depends on Railo's cluster scope and so there is not configuration that can make it ready; either the cluster scope is available or it's not.

**CACHEBOX**

[Home](#) | [Applications](#) | [Agents](#) | [Options](#)

[Reset Cluster Cache](#)
[Reset Server Cache](#)

| Storage Type             | Ready | Description   |
|--------------------------|-------|---|
| CLUSTER                  | No    | In-memory storage using the cluster scope (Railo)               |
| <a href="#">DATABASE</a> | Yes   | Saves cached content to a database table                        |
| DEFAULT                  | Yes   | Standard in-process memory storage                              |
| <a href="#">FILE</a>     | Yes   | Saves cached content to a physical file                         |
| MEMCACHED                | No    | External storage using a memcached server                       |
| NONE                     | Yes   | Disables caching for a specific agent                           |
| SOFT                     | Yes   | Java soft-references allow the JVM to manage content expiration |

| Eviction Policy | Description   |
|-----------------|---|
| AGE             | Expires N minutes after creation                        |
| FIFO            | First In First Out: Expires after N records in cache    |
| IDLE            | Expires after N minutes unused                          |
| LFU             | Least Frequently Used: Expires after N records in cache |
| LRU             | Least Recently Used: Expires after N records in cache   |
| NONE            | Content never expires.                                  |

## Advanced Topics

### Singletons

In order for any caching service to provide an answer for the management of singletons (assuming you want to use the caching system for singleton objects), it must provide a means of addressing the “dog pile” condition. The dog pile is a race condition in which two separate requests (two separate site visitors) attempt to fetch a cached object at the same time, both receive a “miss” from the cache system and proceed to create the object. The reason why this is important is because a singleton object should be unique within an application. So for example if your application has a singleton called the FlightDynamicsOfficer (FIDO) that’s responsible for ensuring safe and successful shuttle launches, you need to be certain that there is one and only one FIDO at run-time. This is the reason why the *STORE* method of the CacheBoxAgent CFC returns the same structure returned by the *FETCH* method. The content value returned from the *STORE* operation will always be the first object created and any other subsequently created objects can be thrown away.

Here is an example of singleton creation using CacheBox.

```
FIDO = agent.FETCH("FIDO");

if (FIDO.status) {
    // a non-zero status is a cache miss, create the object
    objFIDO = CreateObject("component","FIDO").init();
    // now we need to store the object, but...
    // we also need to be sure we have a singleton
    FIDO = agent.STORE("FIDO",objFIDO);
}

// whether we stored or fetched the object,
// it's in the content variable
return FIDO.content;
```

If a request did encounter the race condition and the service attempted to store the singleton twice, then the status value returned from the *STORE* operation will be two (2), indicating a “failure”. This only applies to the storage of objects. If your content is a string like an HTML fragment, then the service will simply overwrite the stored content when new content is provided for the cache name.



## Session Cache

The CacheBox service isn't designed in particular to handle content that might be stored in the session scope. There is however an easy way to store session content in CacheBox. Create an agent in the *APPLICATION* context for session storage. Any time you add content for a session, prefix the CacheName with the user's SessionID. Then when the session expires, delete content from the agent using the wildcard % character to remove all content for the expired session.

```
<cfscript>
// Create Session Agent
Agent = CreateObject("component",
    "cacheboxagent").init("SessionMan");

// Store a session value
Agent.STORE(session.sessionid & ".MyThing");

// Remove all content for an expired session
Agent.DELETE(sessionid & ".%");
</cfscript>
```