

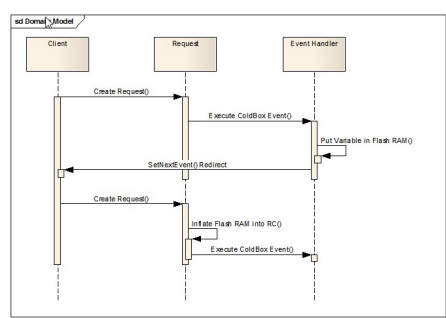
[← Back to Dashboard](#)

ColdBox Flash RAM

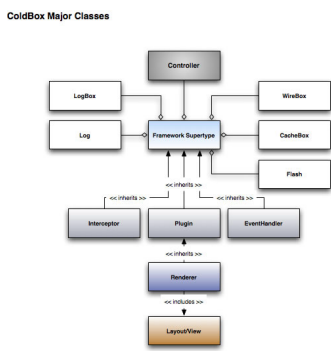
Covers up to version 3.5.0

Introduction

The purpose of the Flash RAM is to allow variables to be persisted seamlessly from one request and be picked up in a subsequent request(s) by the same user. This allows you to hide implementation variables and create web flows or conversations in your ColdBox applications. So why not just use session or client variables? Well, most developers forget to clean them up and sometimes they just end up overtaking huge amounts of RAM and no clean cut definition is found for them. With Flash RAM, you have the facility already provided to you in an abstract and encapsulated format. This way, if you need to change your flows storage scope from session to client scope, the change is seamless and painless.



Every handler, plugin, interceptor, layout and view has a **flash** object in their **variables** scope already configured for usage. The entire flash RAM capabilities are encapsulated in a flash object that you can use in your entire ColdBox code. Not only that, but the ColdBox Flash object is based on an interface and you can build your own Flash RAM implementations very easily. It is extremely flexible to work on a concept of a Flash RAM object than on a storage scope directly. So future changes are easily done at the configuration level.



Our Flash Scope also can help you maintain flows or conversations between requests by using the *discard()* and *keep()* methods. This will continue to expand in our 3.X releases as we build our own conversations DSL to define programmatically wizard like or complex web conversations. Also, the ColdBox Flash RAM has the capability to not only maintain this persistence scope but it also allows you to seamlessly re-inflate these variables into the request collection or private request collection, both or none at all.

Flash Storage

There are times where you need to store user related variables in some kind of permanent storage then relocate the user into another section of your application, be able to retrieve the data, use it and then clean it. All of these tedious operations are definitely double by why relevant the wheel if we can have the platform give us a tool for maintaining conversation variables across requests. The key point for Flash RAM is where will the data be stored so that it is unique per user. ColdFusion gives us several persistent scopes that we can use and we have also created several flash storages for this purpose. Since the ColdBox flash scope is based on an interface, the flash scope storage can be virtually anywhere. You will find all of these implementations in the following package: `coldbox.system.web.flash`. In order to choose what implementation to use in your application you need to tell the [ConfigurationCFC](#) which one to use via **flash** configuration structure:

Configuration

```
// flash scope configuration
Flash = {
  scope = session, client, cluster, ColdboxCache, or full path*
  properties = {} // constructor properties for the flash scope implementation
  inflateToRC true // automatically inflate flash data into the RC scope
  inflateToPRC false // automatically inflate flash data into the PRC scope
  autoPurge true // automatically purge flash data for you
  autoSave true // automatically save flash scopes at end of a request and on relocations.
}
```

Below is a nice chart of all the keys in this configuration structure so you can alter behavior of the Flash RAM objects:

Key	Type	Required	Default	Description
scope	string or instantiation path	false	session	Determines what scope to use for Flash RAM . The available aliases are: session, client, cluster, ColdboxCache or a custom instantiation path
properties	struct	false	{}	Properties that can be used inside the constructor of any Flash RAM implementation
inflateToRC	boolean	false	true	Whatever variables you put into the Flash RAM , they will also be inflated or copied into the request collection for you automatically.
inflateToPRC	boolean	false	false	Whatever variables you put into the Flash RAM , they will also be inflated or copied into the private request collection for you automatically
autoPurge	boolean	false	true	This is what makes the Flash RAM work, it cleans itself for you. Be careful when setting this to false as it then becomes your job to do the cleaning
autoSave	boolean	false	true	The Flash RAM saves itself at the end of requests and on relocations via <i>setNextEvent()</i> . If you do not want auto-saving, then turn it off and make sure you save manually

Core Flash Implementations

The included flash implementations for ColdBox are:

Name	Class	Description
Session	<code>coldbox.system.web.flash.SessionFlash</code>	Persists variables in <i>session</i> scope
Cluster	<code>coldbox.system.web.flash.ClusterFlash</code>	Persists variables in <i>cluster</i> scope via Railo only
Client	<code>coldbox.system.web.flash.ClientFlash</code>	Persists variables in <i>client</i> scope
Mock	<code>coldbox.system.web.flash.MockFlash</code>	Mocks the storage of Flashed variables. Great for unit/integration testing.
ColdboxCache	<code>coldbox.system.web.flash.ColdboxCacheFlash</code>	Persists variables in the ColdBox Cache

Configuration Properties

Each RAM implementation can also use properties in order to alter its behavior upon construction via the **properties** configuration struct. Below are the properties our core implementations can use:

SessionFlash Settings:

- none

ClusterFlash Settings:

- none

ClientFlash Settings:

- none

MockFlash Settings:

- none

ColdboxCacheFlash Settings:

Setting	Type	Required	Default	Description
cacheName	string	false	default	The cache provider name declared in CacheBox to be used to store the user's flash RAM content.

Contents

- [ColdBox Flash RAM](#)
 - [Introduction](#)
 - [Flash Storage](#)
 - [Configuration](#)
 - [Core Flash Implementations](#)
 - [Configuration Properties](#)
 - [Using Flash RAM](#)
 - [Flash Scope Object](#)
 - [clear\(\)](#)
 - [clearFlash\(\)](#)
 - [discard\(\)](#)
 - [exists\(\)](#)
 - [get\(\)](#)
 - [getAccess\(\)](#)
 - [getFlash\(\)](#)
 - [getScope\(\)](#)
 - [isEmpty\(\)](#)
 - [keep\(\)](#)
 - [persistRC\(\)](#)
 - [purge\(\)](#)
 - [purgeAll\(\)](#)
 - [remove\(\)](#)
 - [removeFlash\(\)](#)
 - [saveFlash\(\)](#)
 - [save\(\)](#)
 - [More Examples](#)
 - [Creating Your Own Flash Scope](#)
 - [Implementable Methods](#)
 - [Summary](#)

```
// flash_scope configuration
flash = {
  scope = 'ColdBoxCache'
  properties = { cacheName='master' }
};
```

Using Flash RAM

There are several ways to interact with the ColdBox Flash RAM:

- Using the **flash** scope object (Best Practice)
- Using the **persistVariables()** method from the super type and coldbox controller (*coldbox.system.web.Controller*)
- Using the persistence arguments in the **setNextEvent()** method from the super type and coldbox controller (*coldbox.system.web.Controller*)

All of these methods interact with the Flash RAM object but the last two methods not only place variables in the temporary storage bin but actually serialize the data into the Flash RAM storage immediately. The first approach queues up the variables for serialization and at the end of a request it serializes the variables into the correct storage scope, thus saving precious serialization time. In the next section we will learn what all of this means.

Flash Scope Object

The flash scope object is our best practice approach as it clearly demarcates the code that the developer is using the *flash* scope for persistence. Any flash scope must inherit from our **AbstractFlashScope** and has access to several key methods that we will cover in this section. However, let's start with how the flash scope stores data:

1. The flash persistence methods are called for saving data, the data is stored in an internal temporary request bin and awaiting serialization and persistence either through relocation or termination of the request.
2. If the flash methods are called with immediate save arguments, then the data is immediately serialized and stored in the implementation's persistent storage.
3. If the flash's **saveFlash()** method is called then the data is immediately serialized and stored in the implementation's persistent storage.
4. If the application relocates via **setNextEvent()** or a request finalizes then if there is data in the request bin, it will be serialized and stored in the implementation's storage.

Important: By default the Flash RAM queues up serializations for better performance, but you can alter the behavior programmatically or via the configuration file.

If you use the *persistVariables()* method or any of the persistence arguments on the *setNextEvent()* method, those variables will be saved and persisted immediately.

To review the Flash Scope methods, please [go to the API](#) and look for the correct implementation or the **AbstractFlashScope**. Please note that the majority of a Flash scope methods return itself so you can concatenate method calls. Below are the main methods that you can use to interact with the Flash RAM object:

```
clear()
Clears the temporary storage bin
flash.clear();

clearFlash()
Clears the persistence flash storage implementation
flash.clearFlash();

discard()
any discard([string keys=''])
Discards all or some keys from flash storage. You can use this method to implement flows.
// discard all flash variables
flash.discard();
// discard some keys
flash.discard(userID, userKey, cardID)

exists()
boolean exists(string name)
Checks if a key exists in the flash storage
if( flash.exists('notice') ){
  // do something
}

get()
any get(string name, [any default])
Get's a value from flash storage and you can even pass a default value if it does not exist.
// Get a flash key that you know exists
cardID = flash.get('cardID');
// Render some flash content
<div class="notice">#{flash.get('notice')}</div>

getKeys()
Get a list of all the objects in the temp flash scope.
Flash Keys: #structKeyList( flash.getKeys() )#

getFlash()
Get a reference to the permanent storage implementation of the flash scope.
<cfdump var="#flash.getFlash()">#

getScope()
Get the flash temp request storage used throughout a request until flashed at the end of a request.
<cfdump var="#flash.getScope()">#

isEmpty()
Check if the flash scope is empty or not
<cffif {flash.isEmpty()}
</cffif>

keep()
any keep([string keys=''])
Keep all or a single flash temp variable alive for another relocation. Usually called from interceptors or event handlers to create conversations and flows of data from event to event.
function tep2(event){
  // keep variables for step 2 wizard
  flash.keep(userID, fname, lname)
  // keep all variables
  flash.keep();
}

persistRC()
any persistRC([string include=''], [string exclude=''], [boolean saveNow='false'])
Persist keys from the ColdBox request collection into flash scope. If using exclude, then it will try to persist the entire request collection but excluding certain keys. Including will only include the keys passed from the request collection.
// persist some variables that can be reinflated into the RC upon relocation
flash.persistRC(include=name, email, address)
setNextEvent( wizard.step# )
// persist all RC variables using exclusions that can be reinflated into the RC upon relocation
flash.persistRC(exclude=number);
setNextEvent( wizard.step# )
// persist some variables that can be reinflated into the RC upon relocation and serialize immediately
flash.persistRC(include=email, addressData, venow=TRUE);

put()
any put(string name, any value, [boolean saveNow='false'], [boolean keep='true'], [boolean inflateToRC=FROMConfig], [boolean inflateToPRC=FROMConfig], [boolean autoPurge=FROMConfig])
This is the main method to place data into the flash scope. You can optionally use the arguments to save the flash immediately, inflate to RC or PRC on the next request and if the data should be auto purged for you. You can also use the configuration settings to have a consistent flash experience, but you can most certainly override the defaults. By default all variables placed in flash RAM are automatically purged in the next request once they are inflated UNLESS you use the keep() methods in order to persist them longer or create flows. However, you can also use the autoPurge argument and set it to false so you can control when the variables will be removed from flash RAM. Basically a glorified ColdFusion scope that you can use.
// persist some variables that can be reinflated into the RC upon relocation
flash.put(name=userData, value=userData);
setNextEvent( wizard.step# )
// put and serialize immediately.
flash.put(name=userData, value=userData, saveNow=TRUE);
// put and mark them to be reinflated into the PRC only
flash.put(name=userData, value=userData, inflateToRC=FALSE, inflateToPRC=TRUE);

// put and disable autoPurge, I will remove it when I want to!
flash.put(name=userData, value=userWizard, autoPurge=FALSE);

putAll()
any putAll(struct map, [boolean saveNow='false'], [boolean keep='true'], [boolean inflateToRC='[runtime expression]'], [boolean inflateToPRC='[runtime expression]'], [boolean autoPurge='[runtime expression]'])
Same as the put() method but instead you pass in an entire structure of name-value pairs into the flash scope.
```

```

var map = {
  addressData = rc.address,
  userID = securityService.getUserID(),
  loggedIn = now()
};

// put all the variables in flash scope as single items
flash.putAll(map);

// Use them later
if flash.getLoggedIn(){
}

```

remove()
any remove(string name, [boolean saveNow='false'])
Remove an object from the temporary flash scope so when the flash scope is serialized it will not be serialized. If you would like to remove a key from the flash scope and make sure your changes are reflected in the persistence storage immediately, use the *saveNow* argument.

```

// mark object for removal
flash.remove(notice);

// mark object and remove immediately from flash storage
flash.remove(notice,true);

```

removeFlash()
Remove the entire flash storage. We recommend using the clearing methods instead.

saveFlash()
Save the flash storage immediately. This process looks at the temporary request flash scope and serializes it if it needs to and persists to the correct flash storage on demand.

We would advice to not overuse this method as some storage scopes might have delays and serializations

```
flash.saveFlash();
```

size()
Get the number of the items in flash scope
You have #flash.size()# items in your cart!

More Examples

```

// handler code:
function saveForm(event){
  // save post
  flash.put(notice,"Saved the form baby!");
  // relocate to another event
  setNextEvent(event.show);
}
function show(event){
  // Nothing to do with flash, inflating by flash object automatically
  event.setView(event.show);
}

// User/show.cfm template using if statements
<cfflash,exists?notice>
<div class=notice#flash.get(notice)#</div>
</cfflash>

// User/show.cfm using defaults
<div class=notice#flash.get(name=notice,default=")</div>

```

Creating Your Own Flash Scope

The ColdBox Flash capabilities are very flexible and you can easily create your own Flash Implementations by doing two things:

1. Create a CFC that inherits from `coldbox.system.web.flash.AbstractFlashScope`
2. Implement the following functions: `clearFlash()`, `saveFlash()`, `flashExists()`, and `getFlash()`

Implementable Methods

Method	ReturnType	Description
<code>clearFlash()</code>	void	Will destroy or clear the entire flash storage structure.
<code>saveFlash()</code>	void	Will be called before relocations or on demand in order to flash the storage. This method usually talks to the <code>getScope()</code> method to retrieve the temporary flash variables and then serialize and persist.
<code>flashExists()</code>	boolean	Checks if the flash storage is available and has data in it.
<code>getFlash()</code>	struct	This method needs to return a structure of flash data to reinflate and use during a request.

Important: It is the developer's responsibility to provide consistent storage locking and synchronizations.

All of the methods must be implemented and they have their unique purposes as you read in the description. Let's see a real life example, below you can see the flash implementation for the session scope:

```

<component output=false extends=coldbox.system.web.flash.AbstractFlashScope hint="ColdBox session flash scope">
<----- CONSTRUCTOR ----->
<cfscript>
instance = structnew();
</cfscript>

<--- init --->
<cffunction name=init output=false access=public returnType=sessionFlash hint=Constructor>
<cfargument name=controller type=coldbox.system.web.Controller required=true hint=The ColdBox Controller>
<cfscript>
super.init(arguments.controller);
instance.flashKey=cbox_flash_scope;
return this;
</cfscript>
</cffunction>

<----- IMPLEMENTED METHODS ----->

<--- getFlashKey --->
<cffunction name=getFlashKey output=false access=public returnType=string hint=Get the flash key storage used in session scope.>
<cfreturn instance.flashKey;
</cffunction>

<--- clearFlash --->
<cffunction name=clearFlash output=false access=public returnType=void hint=Clear the flash storage>
<cffunction name=clearFlash output=false access=public returnType=void hint=Clear the flash storage>
<cfif flashExists()>
<set structClear(session[getFlashKey()])
</cfif>
</cffunction>

<--- saveFlash --->
<cffunction name=saveFlash output=false access=public returnType=void hint=Save the flash storage in preparing to go to the next request>
<--- Init The Storage if not Created --->
<cfif NOT flashExists()>
<cflock scope=session throwonTimeout=true timeout=40>
<cfif NOT flashExists()>
<set session[getFlashKey()] = structNew();
</cfif>
</cflock>
</cfif>

<--- Now Save the Storage --->
<set session[getFlashKey()] = getScope();
</cffunction>

<--- flashExists --->
<cffunction name=flashExists output=false access=public returnType=boolean hint=Checks if the flash storage exists and IT HAS DATA to inflate.>
<cfscript>
// Check if session is defined first
if (NOT isDefined(session)) {return false}
// Check if storage is set and not empty
return structKeyExists(session, getFlashKey()) AND NOT structIsEmpty(session[getFlashKey()]);
</cfscript>
</cffunction>

<--- getFlash --->
<cffunction name=getFlash output=false access=public returnType=struct hint=Get the flash storage structure to inflate it.>
<--- Check if Exists, else return empty struct --->
<cfif flashExists()>
<cfreturn session[getFlashKey()];
</cfif>
<cfreturn structnew();
</cffunction>
</component>

```

As you can see from the implementation, it is very straightforward to create a useful session flash RAM object. You can also get more funky and use some ColdBox internal serializers for any kind of object:

```

<component output=false extends=coldbox.system.web.flash.AbstractFlashScope hint="ColdBox client flash scope">
<----- CONSTRUCTOR ----->
<cfscript>

```

```

instance = structnew();
</cfscript>
<!-- init -->
<cffunction name="init" output="false" access="public" returnType="client:Flash" hint="Constructor"
  <cfargument name="controller" type="coldbox.system.web.Controller" required="true" hint="The ColdBox Controller" >
  </cfargument>
  superinit(arguments.controller);

  // Marshall
  instance.converter.createObject(component="coldbox.system.core.conversion.ObjectMarshaller");
  instance.flashKey="box_flash";

  return this;
</cffunction>
</cfscript>
<----- PUBLIC ----->
<!-- getFlashKey -->
<cffunction name="getFlashKey" output="false" access="public" returnType="string" hint="Get the flash key storage used in cluster scope."
  <cfreturn instance.flashKey
</cffunction>
<!-- clearFlash -->
<cffunction name="clearFlash" output="false" access="public" returnType="void" hint="Clear the flash storage"
  <cfif flashExists()
  <cfset structDelete(client, getFlashKey())
  </cfif>
</cffunction>
<!-- saveFlash -->
<cffunction name="saveFlash" output="false" access="public" returnType="void" hint="Save the flash storage in preparing to go to the next request"
  <cfset client[getFlashKey()] = instance.converter.serializeObject( getScope() )
</cffunction>
<!-- flashExists -->
<cffunction name="flashExists" output="false" access="public" returnType="boolean" hint="Checks if the flash storage exists and IT HAS DATA to inflate."
  </cfscript>
  // Check if session is defined first
  if NOT isDefined(client) {return false }
  // Check if storage is set
  return( structKeyExists(client, getFlashKey()) );
</cfscript>
</cffunction>
<!-- getFlash -->
<cffunction name="getFlash" output="false" access="public" returnType="struct" hint="Get the flash storage structure to inflate it."
  <!-- Check if Exists, else return empty struct -->
  <cfif flashExists()
  <cfreturn instance.converter.deserializeObject( client[getFlashKey()] )
  </cfif>

  <cfreturn structnew()
</cffunction>
</cfcomponent>

```

Summary

The ColdBox Flash RAM is an excellent utility to help you create basic or even complex conversations between events and be able to persist data across requests. I really encourage you to start using the Flash RAM capabilities and go funky with them. Expect great things to come as we develop our webflow and conversation DSL language in versions to come.