<< Back to Dashboard

LogBox: The Enterprise ColdFusion Logging Library

Covers up to version 1.6.1

Contents

- LogBox: The Enterprise ColdFusion Logging Library
  - Introduction
    - Some Resources
  - Need for Logging
  - What can LogBox do?
  - How does LogBox work?
    - LogBox
    - Appender
    - Logger
      - Logger Category Inheritance
      - Severity Levels
      - Dynamic Appenders
    - Layout
  - Configuring LogBox
    - Programmatic Configuration
      - LogBox DSL
      - Adding Appenders
      - Configuring the Root Logger
      - Adding Categories To Specific Logging Levels
      - Adding Categories Granularly
    - XML Configuration
    - XML Schema
  - Using LogBox
  - Using a Logger object
    - Instance Members
    - Can Methods For Performance
  - $toString() and ExtraInfo Argument
  - Creating Custom Appenders
    - Helper Methods
    - Instance Members
    - Dealing with Custom Layouts
  - Creating a Custom Layout
    - Instance Members
  - Appender Properties
    - AsyncFileAppender & FileAppender
    - AsyncRollingFileAppender & RollingFileAppender
    - CFAppender
    - ColdBoxTracerAppender
    - DBAppender & AsyncDBAppender
    - EmailAppender
    - ScopeAppender
    - SocketAppender
    - TracerAppender
    - TwitterAppender
  - LogBox in a ColdBox Application
    - Configuration Within ColdBox
      - Programmatic
      - XML
    - Benefits of using LogBox in a ColdBox application
    - Where is LogBox stored in a ColdBox app?
    - LogBox from the ColdBox Factory
    - LogBox from the ColdBox Proxy
    - Can I still use the Logger plugin?
    - The LogBox autowiring DSL
  - Summary

Introduction

LogBox is an enterprise ColdFusion logging library designed to give you flexibility, simplicity and power when logging or tracing is needed in your applications. LogBox is part of the ColdBox Platform 3.0 suite of services and libraries and allows you to easily build upon it's logging framework in order to meet any logging or reporting needs your applications has. LogBox surpasses ColdFusion's very basic *cflog* tag. LogBox allows you to create multiple destinations for your loggings and even configure destinations or change them at runtime.

Almost every application needs logging and/or tracing capabilities and we have developed LogBox to satisfy these needs. Although you should take care not to over-use logging as it can slow down an application, LogBox offers you the capabilities to filter out or cancel logging noise a-la-carte. LogBox was inspired by the original logging capabilities in ColdBox and in the Log4j project.

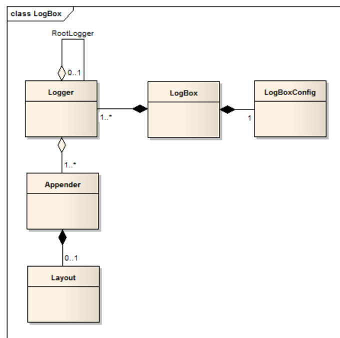
Some Resources

- LogBox Release Notes
- http://logging.apache.org/log4j/

Need for Logging

Most applications require logging and tracing capabilities. One can usually use ColdFusion's standard *cflog* or *cftrace* tags but you can reach a limitation very fast. What if you needed to log only certain severity levels for a particular CFC or piece of code? What if you needed that severity to advise you via SMS or Twitter (yes Twitter)? You would have to build all of these advising and logging capabilities yourself. Also, inserting log statements is tedious, time consuming and frequently pollutes your real code. What if you wanted to turn it off easily or reconfigure your logging levels? You can understand the pain and complexity required to properly deal with these situations. These reasons (and more) are why the ColdBox team has invest in LogBox.

What can LogBox do?



- LogBox can handle the inserting of logging and/or tracing statements in your application with a simple to use API while providing you the ability to manage logging behavior outside of your application code.
- You can configure LogBox via the following means:
  - A programmatic configuration file (cfc).
  - An XML configuration file.
- LogBox categorizes your logging and/or tracing statements according to user-defined categories that can be configured at runtime or pre-runtime. All of these categorizations can have their own logging level ranges (such as *debug* or *info*) and even their own destination points or what we refer to as *LogBox Appenders* (such as *Twitter* or *Console*).
- LogBox Appenders are the destination points you configure for your logging and/or tracing statements. LogBox also offers a basic extensible API so you can build and extend upon the Appender framework according to your unique logging or tracing needs. This gives you complete control and flexibility of how to expand LogBox without reinventing the wheel. Some appenders included in LogBox can log to the following destinations: File, Database, Twitter, Sockets, Email, ColdFusion logging, System Console, and much more.
- LogBox facilitates the creation of your very own customized message formats via *Layouts*. You can create a *Layout* component that can be configured in to *ANY* LogBox appender so it can spit out your very own customized messages.
- LogBox can be instantiated as many times as you want and used as many times as you like in a single application. There are no restrictions upon its usage.
- LogBox allows for category inheritance according to component and package conventions.

How does LogBox work?

LogBox has four main components: *LogBox, Logger, Appenders and Layouts*. These four (4) components work in unison to deliver the logging and tracing of messages and to control how they are logged. You will mostly interact with the *Logger* component as it will send your statements to the Appenders you have configured. Users can extend LogBox and build their own appenders and layouts.

LogBox

LogBox is the core framework you need to instantiate in order to work with logging in your application. You have to instantiate it with a *LogBoxConfig* object that will hold your logging 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 *Logger* object so you can start logging or tracing messages.

You have two ways to use LogBox:

- Standalone Framework
- Within a ColdBox application

If you have downloaded LogBox as a standalone framework, then the initial namespace for the core is *logbox.system*. This allows you to use logbox as a standalone framework that is integrated into your proprietary application.

The ColdBox Framework already has an instance of LogBox created for you in every application and it is stored in the main application controller: *controller.getLogBox()*. The namespace within the ColdBox framework is *coldbox.system*.

Note: Most of the examples shown in this documentation refer the default framework namespace of *coldbox.system*. However, if you are using LogBox as a standalone framework, then the namespace to use is *logbox.system*. ***If you are NOT using the ColdBox Framework, please, take a moment to confirm you are using the correct namespace.***

```
/** Creating LogBox from within a ColdBox Application **/
// Create the config object with an XML configuration file
config = createObject("component","coldbox.system.logging.config.LogBoxConfig").init(expandPath("logbox.xml"));
// Create logbox instance
logBox = createObject("component","coldbox.system.logging.LogBox").init(config);

/** Creating LogBox when used as a stand-alone logging framework **/
// Create the config object with an XML file
config = createObject("component","logbox.system.logging.config.LogBoxConfig").init(expandPath("logbox.xml"));
// Create logbox
logBox = createObject("component","logbox.system.logging.LogBox").init(config);
```

Appender

An *appender* is an object that LogBox uses to log statements to a destination repository. All appenders act as destinations that can include: databases, JMS, Twitter, files, consoles, sockets, etc... The appender has the responsibility of taking the logged message and persisting the message or sending the message to an external service (like Twitter). LogBox comes bundled with the following appenders that can be found in the package *coldbox.system.logging.appenders*:

Appender	Description
AsyncFileAppender	An Asynchronous file appender.
AsyncDBAppender	An Asynchronous database appender.
AsyncRollingFileAppender	An Asynchronous file appender that can do file rotation and archiving.
CFAppender	Will deliver messages to the coldfusion logs.
ColdBoxTracerAppender	Will deliver messages to the ColdBox Tracer Panel in the ColdBox debugger.
ConsoleAppender	Will deliver messages to the server's console via *system.out*
DBAppender	Will deliver messages to a database table. It can auto create the table for you.
EmailAppender	Will deliver messages to any email address.
FileAppender	Will deliver messages a file.
RollingFileAppender	A file appender that can do file rotation and archiving.
ScopeAppender	Will deliver messages to any ColdFusion variable scope.
SocketAppender	Will connect to any server socket and deliver messages.
TwitterAppender	Can either send direct messages to a twitter user or update a status of a twitter user.
TracerAppender	Will deliver messages to the ColdFusion tag *cftrace*.

You can configure LogBox to use one or all of these appenders at any point in time. You can even register as many instances of any appender by defining a unique name for each. Here are examples of how one can configure appenders programmatically or via the simple configuration CFC:

Programmatic Approach

```
//Adding appenders
props = {filePath=expandPath("/coldbox/testing/cases/logging/tmp"),autoExpand=false,fileMaxArchives=1,fileMaxSize=3000};
config.appender(name="MyAsyncFile",class="coldbox.system.logging.appenders.AsyncRollingFileAppender",properties=props);

//twitter
props = {username="myapp",password="",logType="status"};
config.appender(name="TwitterAppender",class="coldbox.system.logging.appenders.TwitterAppender",properties=props);
props = {username="coldbox",password="",logType="dm",dmUser="coldbox"};
config.appender(name="DMTwitter",class="coldbox.system.logging.appenders.TwitterAppender",properties=props);

//socket
props = {host="localhost",timeout="3",port="444",persistConnection=false};
config.appender(name="SocketAppender",class="coldbox.system.logging.appenders.SocketAppender",properties=props);
```

Configuration CFC approach

```
function configure(){

 logBox = {
  // Register Appenders
  appenders = {
   MyAsycFile = {
    class="coldbox.system.logging.appenders.AsyncRollingFileAppender",
    properties={
     filePath=expandPath("/coldbox/testing/cases/logging/tmp"),autoExpand=false,fileMaxArchives=1,fileMaxSize=3000
    }
   },

   TwitterAppender = {
    class="coldbox.system.logging.appenders.TwitterAppender",
    properties = {
     username="myapp",password="",logType="status"
    }
   }

   DMTwitter = {
    class="coldbox.system.logging.appenders.TwitterAppender",
    properties = {
     username="coldbox",password="",logType="dm",dmUser="coldbox"
    }
   },

   SocketAppender = {
    class="coldbox.system.logging.appenders.SocketAppender",
    properties = {
     host="localhost",timeout="3",port="444",persistConnection=false
    }
   }

  }
 };
}
```

Another feature of a LogBox appender is that you can *extend* them or create new ones simply by leveraging the LogBox API. To customize LogBox appenders for your own unique needs you would simply extend the core appender class: *coldbox.system.logging.AbstractAppender* and implementing the *init() and logMessage()* methods. Extending LogBox will be reviewed in greater detail over the next few sections.

Logger

The logger component is a named entity that you use to log or trace messages with. The logger is responsible for deciding if the severity level of the message is adequate for redirecting to a destination via any of its attached appenders. If the logger component deems the message qualifies to be redirected the logger will route the message(s) to the applicable appender(s) for delivery. There are two types of loggers in LogBox:

1. **Root Logger** - The default configured logger **(mandatory)**
2. **Category or Named Loggers** - A logger that is created with a specific category name (Usually a class path). You can define as many as you want.

LogBox requires you to configure a *Root Logger*. A root logger is the default logger that all named category loggers inherit from. This means that if you request a logger by a name you have **NOT** configured before hand, then you will be interacting with the root logger. Configuring the root logger is easy and requires only two sets of instruction: A range of severity levels that the root logger is allowed to respond to; A list of appenders. **Please note, it is a requirement that you configure the root logger.**

```
// Configuring some appenders and the root logger
config.appender(name="Console",class="coldbox.system.logging.appenders.ConsoleAppender");
// root logger
config.root(levelMin=config.logLevels.FATAL, levelMax=config.logLevels.INFO, appenders="Console");

// Or using simple CFC
logBox = {
 appenders = {
  Console = { class="coldbox.system.logging.appenders.ConsoleAppender"}
 },
 root = {levelMin="FATAL", levelMax="INFO", appenders="*"}
}

//Asking logbox for the root logger
Logger = logBox.getRootLogger();
```

Loggers are named entities. The name you provide for a logger is refereed to as the logger's *category* name. This category can be the name of the component or class you are logging messages from or it can be any unique name you desire to distinguish the message being logged.

Important: It is best practice to name your categories with the exact path notation of the component. Ex: *coldbox.system.plugins.BeanFactory*. For simplicity, you can just pass in the object reference and LogBox will figure out the name for you. Example: *logger = logBox.getLogger(this);*

Let's say that I want to log messages in the SES interceptor whose class is *coldbox.system.interceptors.SES*. Then I would do the following to request a logger for usage in my ses interceptor:

```
// A possible approach to defining a category name.
// Declaring a logger while explicitly defining the category name.
logger = logBox.getLogger("coldbox.system.interceptors.SES");
//log an info message
logger.info("Customer deposited fifty billion dollars into your account.");
```

However, you can simplify the code above by passing the instance of where you are logging from. LogBox will then use the object's fully qualified name, via inspection, to define the logger's category name. This approach is simpler and is our preferred approach. Passing *this* will support refactorings and object name changes without burdening your application.

Important: To stay true to our best practice recommendation of passing *this* to the getLogger() method your CFC must have a name attribute within the component declaration. Example:

```
component name="ExampleCFC"{
 ... much removed ...
}


// The prefered approach to defining a category name.
// Declaring a logger while allowing LogBox to implicitly define the category name via introspection.
logger = logBox.getLogger(this);

//log an info message
logger.info("Customer deposited fifty billion dollars into your account. Again.");
```

The above two lines of code illustrate the simplicity of getting a named logger from LogBox. Two important question are: *What severity messages will the logger log?* and *What is the destination (or destinations) of these messages?* In the above example we did not explicitly configure an answer to these two questions while we created a logger for *coldbox.system.interceptors.SES*, so, LogBox will use the *root logger*. Thus it will use the root logger's severity level range and configured appenders. So if we wanted to define a category we can define it in various ways via programmatic approach or XML. Let us now look at the programmatic approach:

```
//register a list of categories that respond to FATAL messages
//only using the root logger's appenders
config.fatal("coldbox.system.controller","mycfc","com.model.myclc");

// log for errors only using the root logger's appenders
config.error("mycfc","com.model.myclc");

//log for info only using the root logger's appenders
config.info("org.model.component");

//register a more granular category with levels and appenders
// Log messages that are from severity 0-1 (fatal - error) only to the twitter appender
config.category(name="com.model.myservice",levelMax=config.logLevel.ERROR,appenders="MyTwit");

//log all email service messages to the MyLogFileAppender and the Console.
config.category(name="org.model.EmailService",appenders="MyLogFileAppender,Console");
```

This is the same but using the simplified Data CFC Approach:

```
logBox = {
 //register a list of categories that respond to FATAL messages
 fatal = ["coldbox.system.controller","mycfc","com.model.myclc"],
 // log for errors only using the root logger's appenders
 error = ["mycfc","com.model.myclc"],
 //log for info only using the root logger's appenders
 info = ["org.model.component"],

 //Register categories granularly:
 categories = {
  "com.model.myservice"={levelMax="ERROR",appenders="MyTwit"},
  "org.model.EmailService"={appenders="MyLogFileAppender,Console"}
 }
};
```

You have the option to create granular categories where you can choose a level range and/or appenders, or easily tag a category to listen to only one specific type of message using the severity methods in the config object. The methods available are the same as the **Severity** column in the severity levels chart.

Logger Category Inheritance

We have a *convention* for our category names where each category name uses dot-notation according to the components path. Using a class-path convention for our category names allows one to pseudo-inherit for logging levels and appenders! The following example should help clarify this concept.

The overall premise is that when you request a logger with a category name LogBox will search for it's configuration. If LogBox does not find a configuration for the category name it will try to locate its closest ancestor for logging levels and appenders. If LogBox cannot find an ancestor the message will be logged using the *root logger* information. For example, let's say we define some categories like this:

```
// Configured Appenders: FileAppender, ConsoleAppender, TwitterAppender
config.category(name="coldbox.system",levelMin=config.logLevels.INFO,appenders="console");
config.error("coldbox.system.plugins");
```

Then, let's say we request the following logger objects:

```
logger = logBox.getLogger("coldbox.system.plugins.BeanFactory");
logger.info("hello info");
logger.error("wow and error occurred");

logger = logBox.getLogger("coldbox.system.interceptors.SES");
logger.info("hello info");
logger.debug("a cool debug message");
```

Note: All example code snippets are using a *getLogger('categoryname')* call instead of our preferred approach of *getLogger(this)* because we want to showcase which category we are talking about. Please take this into consideration.

Category	Configured Levels	Assigned Levels	Appenders
root	FATAL-DEBUG	FATAL-DEBUG	console,file,twitter
coldbox.system	INFO-DEBUG	INFO-DEBUG	console
coldbox.system.plugins	ERROR	ERROR	*
coldbox.system.interceptors.SES	NONE	INFO-DEBUG from *coldbox.system*	console from *coldbox.system*

coldbox.system.plugins.BeanFactory	NONE	ERROR from *coldbox.system.plugins*	*

Since we requested the category: *coldbox.system.plugins.BeanFactory*, LogBox tries to locate it, but it has not been defined, so it takes off the last item in the category name. Now it will search for a category of: *coldbox.system.plugins* via pseudo-inheritance. However, now *coldbox.system.plugins* has been found and it has been configured to only listen to *error* messages. Therefore, the *coldbox.system.plugins.BeanFactory* logger can ONLY log error messages according to it's inherited category. So the *info()* message will be ignored!

The second logger is called *coldbox.system.interceptors.SES*, LogBox tries to match a category but fails, so it now searches for a logger called *coldbox.system.interceptors*. It still cannot find it so it continues up the package chain and finds the *coldbox.system* logger which has been set with a minimum of *DEBUG* level and ONLY the console appender. So the only message that gets logged is the *logger.debug()* message and it will be sent to the console appender.

These examples should give you insight into category inheritance and the power they provide. You can easily turn toggle logging for entire packages with a single category definition. However, this is great only if you follow the dot notation conventions. Below is a sample generic chart sample:

Category	Configured Levels	Assigned Levels
root	FATAL-DEBUG	FATAL-DEBUG
x	NONE	FATAL-DEBUG from root
x.y	INFO	INFO
x.y.z	NONE	INFO from x.y

Severity Levels

Each logger will be configured with an optional severity level range: *LevelMin* and *LevelMax*. These severities are integers from -1 to 4, each representing a severity:

Severity	Integer Level
OFF	-1
FATAL	0 (Default LevelMin)
ERROR	1
WARN	2
INFO	3
DEBUG	4 (Default LevelMax)

As you can see from the chart above the default minimum level is *FATAL* (all messages are ignored except fatal, uncaught errors) and the highest level is *DEBUG* (debug listens and logs all messages). When you define a root logger or category logger you should define them using a severity level from the table above. If you do not define a severity level a default level will be used. Once you have a logger instantiated you can dynamically change the logging levels by interacting with the *setLevelMin() and setLevelMax()* methods of the logger you have instantiated.

```
//change min level of logging to warn only
logger = logBox.getRootLogger();
logger.setLevelMin(logger.logLevels.WARN);
```

Each logger object has a public property called *logLevels* that maps to the component: *coldbox.system.loggging.LogLevels* which is used as a static lookup of severity levels. You may use the alias, or, the numeric level; however, it is best practice to use a level alias (such as DEBUG or INFO).

Dynamic Appenders

Each logger object has several methods that you can use in order to interact with the logger's appenders. You can add, remove, clear or list the appenders on a specific logger instance. Below are the methods you can use in the logger class to interact with appenders:

Method	Return Type	Description
hasAppenders()	*Boolean*	Checks if the logger has any appenders attached to it
getAppenders()	*Struct*	Returns the map of registered appenders
getAppender(name)	*Appender*	Return a named appender if it is registered in the logger
appenderExists(name)	*Boolean*	Checks if a named appender exists in the logger
addAppender(Appender)	*void*	Register an appender with the logger at runtime
removeAppender(name)	*Boolean*	Will un-register an appender from this logger
removeAllAppenders()	*void*	Will try to un-register all appenders from this logger

So you can easily add/remove/check the appenders on any logger at any time.

```
//Add your own appender at runtime
jms = createObject("component","com.appender.JMSAppender").init("JMSAppender",properties);
logger.addAppender(jms);

//log a message to all appenders and to my jms appender:
logger.fatal("I FAILED MAN!");

//remove it
logger.removeAppender("JMSAppender");
```

Layout

The layout component defines the format of the message to store in an appender repository. Be default, each appender already has a pre-defined message format. However, if you do not like the format of the message you can easily change it by creating your own *layout* component and registering it with the appender. You can do this in the configuration object when you add appenders:

```
//add a FileAppender with my own formatting
props = {filePath='/logs',fileName='Test'};
config.appender(name='FileApp',
       class="coldbox.system.logging.appenders.FileAppender",
    properties=props,
       layout="model.logging.MyFileLayout");
```

So to create your very own *layout* object, you just need to extend the LogBox abstract layout object: *coldbox.system.logging.Layout*.

Configuring LogBox

We already have a taste of how to configure LogBox, but let's go into details. There are three approaches to configuring LogBox: two programmatic approaches or an XML approach. We definitely lean towards our programmatic approach as it provides much more flexibility and less verbosity. So let's cover it first.

Programmatic Configuration

No matter what configuration you decide to use, you will always have to instantiate LogBox with a LogBoxConfig object: *coldbox.system.logging.config.LogBoxConfig*. However you have the option of either talking directly to this CFC or creating a more portable configuration. This portable configuration we denote as a simple data CFC that contains the LogBox configuration data using what we call our LogBox DSL (Domain Specific Language). The cool thing about this LogBox DSL is that it is exactly the same whether you are using LogBox in ColdBox applications or in any other framework or non-framework ColdFusion application. So you can configure LogBox by:

1. Creating a portabla data CFC using the LogBox DSL or
2. Creating the LogBoxConfig object and interacting with its methods

LogBox DSL

In order to create a simple data CFC, just create a CFC with one required method on it called *configure* where you will define the logging configuration data:

```
**
* A LogBox configuration data object
*/
component{

 function configure(){
  logBox = {

  };
 }
}
```

Once you have this shell, you will create a *logBox* variable in the *variables* scope that must be a structure with the following keys:

Key	Description
appenders	A structure where you will define appenders
root	A structure where you will configure the root logger
categories	A structure where you can define granular categories (OPTIONAL)
DEBUG	An array that will hold all the category names to place under the DEBUG logging level (OPTIONAL)
INFO	An array that will hold all the category names to place under the INFO logging level (OPTIONAL)
WARN	An array that will hold all the category names to place under the WARN logging level (OPTIONAL)
ERROR	An array that will hold all the category names to place under the ERROR logging level (OPTIONAL)
FATAL	An array that will hold all the category names to place under the FATAL logging level (OPTIONAL)
OFF	An array that will hold all the category names to not log at all (OPTIONAL)

So to define an appender you must define a key value which is the internal name of the appender with the following keys:

- **class** : The class path of the appender
- **layout** : The layout class path of the layout object to use (optional)
- **properties** : The structure of name-value pairs to configure this appender with (optional)
- **levelMin** : The numerical or English word of the minimal logging level (optional, defaults to 0)
- **levelMax** : The numerical or English word of the maximum logging level (optional, defaults to 4)

To define the root logger you can use the following keys:

- **levelMin** : The numerical or English word of the minimal logging level (optional, defaults to 0)
- **levelMax** : The numerical or English word of the maximum logging level (optional, defaults to 4)
- **appenders** *: A string list of the appenders to use for logging*

To define categories you can use the following keys in the *categories* structure and the key of the structure is the name of the category:

- **levelMin** : The numerical or English word of the minimal logging level (optional, defaults to 0)
- **levelMax** : The numerical or English word of the maximum logging level (optional, defaults to 4)
- **appenders** *: A string list of the appenders to use for logging (optional, defaults to *)*

As you might notice the name of the keys on all the structures match 100% to the programmatic methods you can also use to configure logBox. So when in doubt, refer back to the argument names.

```
logBox = {
 // Appenders
 appenders = {
  appenderName = {
   class="class.to.appender",
   layout="class.to.layout",
   levelMin=0,
   levelMax=4,
   properties={
    name  = value,
    prop2 = value 2
   }
 },
 // Root Logger
 root = {levelMin="FATAL", levelMax="DEBUG", appenders="*"},
 // Granualr Categories
 categories = {
  "coldbox.system" = { levelMin="FATAL", levelMax="INFO", appenders="*"},
  "model.security" = { levelMax="DEBUG", appenders="console,twitter"}
 }
 // Implicit categories
 debug  = ["coldbox.system.interceptors"],
 info   = ["model.class", "model2.class2"],
 warn   = ["model.class", "model2.class2"],
 error  = ["model.class", "model2.class2"],
 fatal  = ["model.class", "model2.class2"],
 off    = ["model.class", "model2.class2"]
};
```

Once you have defined the configuration data in this object you can now use the same LogBox Config object to either instantiate it for you or you can pass a reference of it by using the *init()* method of the *LogBoxConfig* object:

```
init([XMLConfig,CFCConfig,CFCConfigPath])
```

- **XMLConfig** : The absolute path of the logbox XML configuration file
- **CFCConfig** : The object instance that has the logbox configuration data
- **CFCConfigPath** : The instantiation path of the object that has the logbox configuration data

```
// Using config path
config = createObject("component","coldbox.system.logging.config.LogBoxConfig").init(CFCConfigPath="my.path.LogBoxConfig");
logBox = createObject("component","coldbox.system.logging.LogBox").init(config);

//Using config object
data   = createObject("component","my.data.CFC");
config = createObject("component","coldbox.system.logging.config.LogBoxConfig").init(data);
logBox = createObject("component","coldbox.system.logging.LogBox").init(config);
```

That's it! Using this DSL approach, your configurations are much more portable now and can even be shared in ANY framework, ColdBox or ColdFusion application. So now let's explore how to bypass this data CFC and use the *LogBoxConfig* object directly. It is important to understand these methods as they are called for you when you define your LogBox DSL data.

Adding Appenders

The first thing you need to do in your config object is add appenders. Each appender is added via the *appender()* method.

public void appender(string name, string class, [struct properties], string layout=)

Parameters:

- **name** - A unique name for the appender to register. Only unique names can be registered per instance.
- **class** - The appender's class to register. We will create, init it and register it for you.
- **properties** - The structure of properties to configure this appender with.
- **layout** - The layout class path to use in this appender for custom message rendering.
- **levelMin** : The numerical or English word of the minimal logging level (optional, defaults to 0)
- **levelMax** : The numerical or English word of the maximum logging level (optional, defaults to 4)

```
config.appender(name="CFConsole",class="coldbox.system.logging.appenders.ConsoleAppender");
config.appender(name="MyCF",class="coldbox.system.logging.appenders.CFAppender");

props = {host="localhost",timeout="3",port="444",persistConnection=false};
config.appender(name="SocketBaby",class="coldbox.system.logging.appenders.SocketAppender",properties=props);

props = {filePath="/logs",fileName="Test"};
config.appender(name="Fileapp",
       class="coldbox.system.logging.appenders.FileAppender",
    properties=props,
       layout="model.logging.MyFileLayout");
```

Configuring the Root Logger

This is also mandatory if you will be using LogBox, you must add a root logger configuration. This is very easy and few arguments.

public void root([numeric levelMin='-1'], [numeric levelMax='4'], string appenders)

Parameters:

- **levelMin** - The default log level for the root logger, by default it is 0. Optional.
- **levelMax** - The default log level for the root logger, by default it is 4. Optional.
- **appenders** - A list of appenders to configure the root logger with. Use * to add all registered appenders

```
config.root(appenders="*");
config.root(levelMax=config.logLevels.WARN,appenders="console,files");
config.root(levelMin=config.logLevels.INFO,levelMax=config.logLevels.DEBUG,appenders="*");
```

Adding Categories To Specific Logging Levels

The methods shown below are used to add categories to specific severity levels only. Each method can receive 1 to* category arguments.

- *public void DEBUG()*
- *public void INFO()*
- *public void WARN()*
- *public void ERROR()*
- *public void FATAL()*
- *public void OFF()*

```
config.debug("com.model.myclass","coldbox.system.controller");
config.info("com.model.otherclass","coldbox.system.whatever");
config.fatal("com.model.otherclass","coldbox.system.whatever");
config.error("com.model.otherclass","coldbox.system.whatever");
config.OFF("com.model.otherclass","coldbox.system.whatever");
```

Adding Categories Granularly

You can also add categories with very granular information using the *category()* method. This method will allow you to add a category definition with a range of severity levels and even a list of which appenders to respond to.

public void category(string name, [numeric levelMin='0'], [numeric levelMax='4'], [string appenders=''])*

Parameters:

- **name** - A unique name for the appender to register. Only unique names can be registered per instance.
- **levelMin** - The default min log level for this category. Defaults to the lowest level 0 or FATAL
- **levelMax** - The max default log level for this category. If not passed it defaults to the highest level possible
- **appenders** - A list of appender names to configure this category with else it will use all the appenders in the root logger. You can also use * to add all registered apenders.

```
//register a more granular category with levels and appenders
// Log messages that are from severity 0-1 (fatal - error) only to the twitter appender
config.category(name="com.model.myservice",levelMax=config.logLevel.ERROR,appenders="MyTwit");

//log all email service messages to the MyLogFileAppender and the Console.
config.category(name="org.model.EmailService",appenders="MyLogFileAppender,Console");
```

XML Configuration

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

```
config = createObject("component","coldbox.system.logging.config.LogBoxConfig").init(expandPath("/config/logbox.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().getLogBoxConfig();
//create the log box config object
config = createObject("component","coldbox.system.logging.config.LogBoxConfig").init();
config.parseAndLoad(xmlDoc);
```

Sample *logbox.xml* file:

```
<?xml version="1.0" encoding="UTF-8"?>
<LogBox xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://www.coldbox.org/schema/LogBoxConfig_1.4.xsd">

 <-- <Appender Definitions -->
 <Appender name="myconsole" class="coldbox.system.logging.appenders.ConsoleAppender" />
 <Appender name="MyCF" class="coldbox.system.logging.appenders.CFAppender" layout="model.myLayout">
  <Property name="fileName">MyAppLogName</Property>
 </Appender>
 <Appender name="FileAppender" class="coldbox.system.logging.appenders.AsyncRollingFileAppender" levelMax="DEBUG">
  <Property name="filePath">/coldbox/testing/logging/tmp</property>
  <Property name="fileMaxSize">3</Property>
```

```
  <Property name="fileMaxArchives">2</Property>
 </Appender>

 <-- <Root Logger -->
  <-- <Root All Appenders
   <Root levelMin="0" levelMax="4" appenders="*">
 -->
 <Root levelMin="0" levelMax="4">
  <Appender-ref ref="myconsole"/>
  <Appender-ref ref="MyCF"/>
  <Appender-ref ref="FileAppender"/>
 </Root>

 <-- <Very advanced category -->
 <Category name="MySES" levelMin="3" levelMax="4">
  <Appender-ref ref="myconsole"/>
 </Category>

 <Category name="com.model.services" levelMax="3" appenders="MyCF"/>
 <Category name="com.model.dao" levelMax="2" appenders="*" />
</LogBox>
```

As you can see, you need to create a root element called *LogBox* with the following child elements:

- *<Appender>* : an appender definition. Can be 1 or more elements.
  - *@name* : The name of the appender (required)
  - *@class* : The class of the appender (required)
  - *@layout* : The layout class to use for rendering the messages (optional)
  - *@levelMin* : The default minimum log level. (optional)
  - *@levelMax* : The default maximum log level. (optional)
- *<Root>* : The root logger element. Can only be 1 defined.
  - *@levelMin* : The default minimum log level. (optional)
  - *@levelMax* : The default maximum log level. (optional)
  - *@appenders* : An optional list of appenders for the root logger or* for all appenders. (optional)
  - *<Appender-ref>* : If you do not like to use a list for the appenders, you can use this element to add appenders to the root logger. You can have 0 or more of these elements defined.
    - *@ref* : The name of the appender to reference.
- *<Category>* : A category definition element. You can have 0 or more of these defined.
  - *@name* : The name of the category
  - *@levelMin* : The default minimum log level. (optional)
  - *@levelMax* : The default maximum log level. (optional)
  - *@appenders* : An optional list of appenders for this category or* for all appenders. (optional)
  - *<Appender-ref>* : If you do not like to use a list for the appenders, you can use this element to add appenders to the category. You can have 0 or more of these elements defined.
    - *@ref* : The name of the appender to reference.

Note: All the *levelMin and levelMax* attributes can either be the numeric representation of the severity or the fully qualified name you can see in the severity table.

XML Schema

We have also included a schema fileL *coldbox/logging/config/LogBoxConfig.xsd* that you can use to validate your XML and use cool tag insight and introspection. You can add the following header to your XML declaration file to enable it:

```
<?xml version="1.0" encoding="UTF-8"?>
<LogBox xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://www.coldbox.org/schema/LogBoxConfig_1.4.xsd">

</LogBox>
```

If your IDE supports XML introspection, this will do the trick.

Using LogBox

Once you have created and configured the LogBox library, you can interact with it in order to get logger objects. The main methods you will use to interact with LogBox are the following, but I recommend you look at the CFC api (http://www.coldbox.org/api) in order to get a listing of all available methods.

- *LogBoxConfig getConfig()* : Get the config object registered
- *Logger getLogger(any category)* : Get a named logger object using a category string or the object you will log from
- *Logger getRootLogger()* : Get a reference to the root logger
- *string getVersion()* : Get the current version of LogBox
- *string getCurrentAppenders()* : Get a list of currently registered appenders
- *string getCurrentLoggers()* : Get a list of currently instantiated loggers
- *void configure(LogBoxConfig config)* : Dynamically re-configure the LogBox library

The two most important methods are *getRootLogger() & getLogger()*, which you will use to get the root or named logger objects.

Important: When you ask for a named category logger and LogBox cannot find its definition, it will create a logger that will inherit its logging levels and appenders from the root logger.

Using a Logger object

Once you retrieve a logger object from LogBox, you are ready to start sending messages. We already covered how to dynamically add/remove/list/check appenders from a logger, so let's look at the other methods we have available:

Utility Methods

- *boolean canLog(numeric level)* : Checks if this logger can log a certain type of severity, there is also a *can{severity}()* method for each severity level.
- *boolean can{severity}()* : Checks if this logger can log a certain type of severity
- *string getCategory()* : Get this logger's category name
- *void setCategory(category)* : Set the category name
- *Logger getRootLogger()* : Get the root Logger
- *numeric getLevelMin()* : Get the minimum severity level
- *void setLevelMin(numeric level)* : Set the minimum severity level
- *numeric getLevelMax()* : Get the maximum severity level
- *void setLevelMax(numeric level)* : Set the maximum severity level

Logging Methods

- *fatal(string message, [any extraInfo=' '])* : Log a fatal message
- *error(string message, [any extraInfo=' '])* : Log an error message
- *warn(string message, [any extraInfo=' '])* : Log a warning message
- *info(string message, [any extraInfo=' '])* : Log an information message
- *debug(string message, [any extraInfo=' '])* : Log a debug message
- *logMessage(string message, numeric severity, [any extraInfo=' '])* : Log any kind of message

As you can probably tell, all logging methods take in a message string an a second argument called *extraInfo*. This *extraInfo* argument can be anything from a string, a structure, a query or whatever. This way you can send in a complex structure that the appenders will serialize into message form or log into its appropriate channel. Thus, *extraInfo* can be very handy when you are building your own custom appenders.

```
// setting some messages
myLogger = logBox.getLogger(this); //"com.model.dao"

myLogger.info("I just created my first logger");

try{
 data = dao.getDBData();
}
catch(Any e){
 myLogger.error("Something really died on my dbdata method: #e.message# #e.detail#", e.tagContext);
}
```

I hope that by now you understand the basics of loggers and how easy it is to use them.

Instance Members

Every logger has access to the following public variables:

- *this.logLevel* : A reference to the *coldbox.system.logging.LogLevels* class

Can Methods For Performance

We recommend using the available *can{severity}()* methods to determine if we can log at a specific log level before actually writing the logging method line. This is done as best practice in order to avoid processing of messages that will never be logged anyways. So let's look at a very simple example of what **NOT** to do:

```
log.debug("This is my log message, some #dynamic# date, i have", dataCFC);
```

This will call the logger's *debug()* method, execute the lines of code and then the logger determines if it can be logged or not. This is ok, but we all love performance and best practice, so we encourage you to do the following:

```
if( log.canDebug() ){
 log.debug("This is my log message, some #dynamic# date, i have", dataCFC);
}
```

This way, the logger determines if it can send debug log messages, and only IF IT CAN, it does so. This is much more faster and cleaner, but you will type more, sorry!

$toString() and ExtraInfo Argument

When using any of the logging methods like *info(),debug(),warn(), etc*, they all take two arguments: 1) The message to log and 2) An extraInfo argument which can be anything you like. This extra info argument can be a simple value, a CFC, a complex object and pretty much anything you like. The appenders get this extra info argument and they can process it into their appropriate destinations by serializing its value to simple form. This is done by using the following algorithm:

1. If it is a simple value, then just use it
2. If it is an object then check if the object has a method called *$toString()*
   1. If the method exists, then call *$toString()* and use its return value
3. If it is an object with NO *$toString()* then marshall its representation in XML format
4. If it is a complex variable like a struct, query, array, etc, then marshall it to JSON format

As you can see from the algorithm above, you can use the extra info argument to your benefit to save serialized representations of data to the appenders and then retrieve or re-inflate them later. The *$toString()* convention is great because you have complete control on how a CFC will serialize to its string representation. Let's see an example on a simple CFC:

```
component{

 function $toString(){
  // return my representation as a comma list of values of my properties
  return "#getName()#,#getAge()#,#getEmail()#";
 }
}
```

So when this object is sent to a logger's method, it will detect it is an object and the *$toString()* function exists and call it for serialization.

```
user = userService.getUser(rc.id);

// need to log it.
if( log.canDebug() ){
 log.debug("User just got logged in right now!", user);
}
```

Creating Custom Appenders

In order to create your own appenders, you will have to create a cfc that extends *coldbox.system.logging.AbstractAppender* and implement the following methods:

- *init()* : Your constructor

- *logMessage()* : The method that is called when a message is received
- *onRegistration()* : An interceptor that fires when the appender gets created and initialized. It can be used for preparing the appender for operation.
- *onUnRegistration()* : An interceptor that fires when the appender is removed from a logger.

The signature of the init method is the following:

```
<!--- Init --->
<cffunction name="init" access="public" returntype="AbstractAppender" hint="Constructor called by a Concrete Appender" output="false">
 <!--- ************************************************************* --->
 <cfargument name="name"  type="string" required="true" hint="The unique name for this appender."/>
 <cfargument name="properties" type="struct" required="false" default="#structnew()#" hint="A map of configuration properties for the appender"/>
 <cfargument name="layout"  type="string" required="false" default="" hint="The layout class to use in this appender for custom message rendering."/>
 <cfargument name="levelMin"  type="numeric" required="false" default="0" hint="The default log level for this appender, by default it is 0. Optional. ex: LogBox.logLevels.WARN"/>
 <cfargument name="levelMax"  type="numeric" required="false" default="4" hint="The default log level for this appender, by default it is 5. Optional. ex: LogBox.logLevels.WARN"/>
 <!--- ************************************************************* --->

</cffunction>
```

As you can see each appender receives a name, a structure of properties, a layout class, an optional levelMin and levelMax severity levels. The properties and layout are both optional, but you *must* call the *super.init()* method in order to have full ok operation on the appender. You can then do your own constructor as you see fit. Here is an example:

```
<!--- Constructor --->
<cffunction name="init" access="public" returntype="FileAppender" hint="Constructor" output="false">
 <!---************************************************************* --->
 <cfargument name="name"  type="string" required="true" hint="The unique name for this appender."/>
 <cfargument name="properties" type="struct" required="false" default="#structnew()#" hint="A map of configuration properties for the appender"/>
 <cfargument name="layout" type="string" required="true" default="" hint="The layout class to use in this appender for custom message rendering."/>
 <cfargument name="levelMin"  type="numeric" required="false" default="0" hint="The default log level for this appender, by default it is 0. Optional. ex: LogBox.logLevels.WARN"/>
 <cfargument name="levelMax"  type="numeric" required="false" default="4" hint="The default log level for this appender, by default it is 5. Optional. ex: LogBox.logLevels.WARN"/>
     <!---************************************************************* --->
 <cfscript>
  super.init(argumentCollection=arguments);

  // Setup Properties
  if( NOT propertyExists("filepath") ){
   $throw(message="Filepath property not defined",type="FileAppender.PropertyNotFound");
  }
  if( NOT propertyExists("autoExpand") ){
   setProperty("autoExpand",true);
  }
  if( NOT propertyExists("filename") ){
   setProperty("filename",getName());
  }
  if( NOT propertyExists("fileEncoding") ){
   setProperty("fileEncoding","UTF-8");
  }

  // Setup the log file full path
  instance.logFullpath = getProperty("filePath");
  // Clean ending slash
  if( right(instance.logFullpath,1) eq "/" OR right(instance.logFullPath,1) eq "\" ){
   instance.logFullPath = left(instance.logFullpath, len(instance.logFullPath)-1);
  }
  instance.logFullPath = instance.logFullpath & "/" & getProperty("filename") & ".log";

  // Do we expand the path?
  if( getProperty("autoExpand") ){
   instance.logFullPath = expandPath(instance.logFullpath);
  }

  //lock information
  instance.lockName = getname() & "logOperation";
  instance.lockTimeout = 25;

  return this;
 </cfscript>
</cffunction>
```

The signature of the *logMessage* method is the following:

```
<!--- logMessage --->
<cffunction name="logMessage" access="public" output="false" returntype="void">
 <!---************************************************************* --->
 <cfargument name="logEvent" type="coldbox.system.logging.LogEvent" required="true" hint="The logging event to log.">
 <!---************************************************************* --->

</cffunction>
```

As you can see it is a very simple method that receives a LogBox logging event object. This object keeps track of the following properties with its appropriate getters and setters:

- timestamp
- category
- message
- severity
- extraInfo

You can then use this logging event object to log to whatever destination you want. Here is a snippet from our scope appender:

```
<!--- Log Message --->
<cffunction name="logMessage" access="public" output="true" returntype="void" hint="Write an entry into the appender.">
 <!---************************************************************* --->
 <cfargument name="logEvent" type="coldbox.system.logging.LogEvent" required="true" hint="The logging event"/>
 <!---************************************************************* --->
 <cfscript>
  var logStack = "";
  var entry = structnew();
  var limit = getProperty("limit");
  var loge = arguments.logEvent;

  // Verify storage
  ensureStorage();

  // Check Limits
  logStack = getStorage();

  if( limit GT 0 and arrayLen(logStack) GTE limit ){
   // pop one out, the oldest
   arrayDeleteAt(logStack,1);
  }

  // Log Away
  entry.id = createUUID();
  entry.logDate = loge.getTimeStamp();
  entry.appenderName = getName();
  entry.severity = severityToString(loge.getseverity());
  entry.message = loge.getMessage();
  entry.extraInfo = loge.getextraInfo();
  entry.category = loge.getCategory();

  // Save Storage
  arrayAppend(logStack, entry);
  saveStorage(logStack);
 </cfscript>
</cffunction>
```

Finally, both the *onRegistration and onUnRegistration* methods have to be *void* methods with no arguments.

```
<cffunction name="onRegistration" access="public" hint="Runs after the appender has been created and registered. Implemented by Concrete appender" output="false" returntype="void">
</cffunction>

<cffunction name="onUnRegistration" access="public" hint="Runs before the appender is unregistered from LogBox. Implemented by Concrete appender" output="false" returntype="void">
</cffunction>
```

These are great for starting or stopping your appenders if they so need to. Here is a sample from our socket appender:

```
<!--- onRegistration --->
<cffunction name="onRegistration" output="false" access="public" returntype="void" hint="When registration occurs">
 <cfif getProperty("persistConnection")>
  <cfset openConnection()>
 </cfif>
</cffunction>

<!--- onRegistration --->
<cffunction name="onUnRegistration" output="false" access="public" returntype="void" hint="When Unregistration occurs">
 <cfif getProperty("persistConnection")>
  <cfset closeConnection()>
 </cfif>
</cffunction>
```

Helper Methods

The abstract appender also has various cool methods that you can use when building appenders:

CF Utility Methods

- *$abort()*
- *$dump(any var, boolean isAbort=[false])*
- *$log(string severity='INFO', string message=) : Log a cflog message just in case*
- *$rethrowit(any throwObject)*
- *$throw(string message, [string detail=*], [string type='Framework'])

Properties Methods

- *struct getProperties()* : Get all the properties struct
- *void setProperties(struct properties)* : Override all properties
- *any getProperty(string property)* : Get a property
- *void setProperty(string property, any value)* : Set a property
- *Boolean propertyExists(string property)* : Checks if a property exists

Utility Methods

- *isInitialized()* : If the appender has been initialized
- *getName()* : Get the name of the appender
- *getHash()* : Get the appender's unique hash id
- *severityToString(numeric severity)* : Transforms a severity integer to it's human readable form

Layout Methods

- *any getCustomLayout()* : Get the custom layout object if defined.
- *boolean hasCustomLayout()* : Checks if the custom layout object is defined in this appender.

Instance Members

Every *Appender* has access to the following public variables:

- *this.logLevel* : A reference to the *coldbox.system.logging.LogLevels* class

Dealing with Custom Layouts

In order for an appender to deal with custom layouts, you must use the layout methods when preparing to log your messages. Below is a simple example from the console appender of how to do this:

```
if( hasCustomLayout() ){
  entry = getCustomLayout().format(loge);
}
else{
  entry = "#severityToString(loge.getseverity())# #loge.getCategory()# #loge.getmessage()# ExtraInfo: #loge.getextraInfoAsString()#"
}

// Log message to system.out
instance.out.println(entry);
```

As you can see, all you need to do is have an if statement that checks whether the appender has a custom layout or not and then assign the return of the layout as your message to log.

Creating a Custom Layout

You can easily create a custom layout object by creating a cfc that extends our abstract layout object: *coldbox.system.logging.Layout* and implementing a *format()* method. Below you can see the method signature:

```
<--- format --->
<cffunction name="format" output="false" access="public" returntype="string" hint="Format a logging event message into your own format">
 <cfargument name="logEvent" type="coldbox.system.logging.LogEvent" required="true"  hint="The logging event to use to create a message.">

</cffunction>
```

All you need to do is inspect the logging event and create your very own message and then return it back. That's it! You thought there was more?

Instance Members

Every *Layout* has access to the following public variables:

- *this.logLevel* : A reference to the *coldbox.system.logging.LogLevels* class
- *this.LINE_SEP* : A line separator equal to *chr(13) & chr(10)*

Appender Properties

As we mentioned before, LogBox ships with over 10 different appenders for your logging and tracing needs. Some of them require configuration properties and some don't. We already discovered that when we configure an appender we can pass in a structure of properties much like how we configure ColdBox Interceptors. Each appender can implement as many properties as they see fit. Below we will digest all the included LogBox appenders and their configuration properties.

AsyncFileAppender & FileAppender

Property	Type	Required	Default	Description
filePath	string	true	---	The location of where to store the log file
filename	string	false	Name of the Appender	The name of the file, if not defined, then it will use the name of this appender. Do not append an extension to it. We will append a *.log* to it
fileEncoding	string	false	utf-8	The file encoding to use, by default we use UTF-8
autoExpand	boolean	false	true	Whether to expand the file path or not. Defaults to true

Note: Please remember to set the *autoExpand* property to FALSE if you will be using an absolute file path location.

AsyncRollingFileAppender & RollingFileAppender

Property	Type	Required	Default	Description
filePath	string	true	---	The location of where to store the log file
filename	string	false	Name of the Appender	The name of the file, if not defined, then it will use the name of this appender. Do not append an extension to it. We will append a *.log* to it
fileEncoding	string	false	utf-8	The file encoding to use, by default we use UTF-8
autoExpand	boolean	false	true	Whether to expand the file path or not. Defaults to true
fileMaxSize	int	false	2000 (2MB)	The max file size for log files. Defaults to 2000 (2 MB)
fileMaxArchives	int	false	2	The max number of archives to keep. Defaults to 2

Note: Please remember to set the *autoExpand* property to FALSE if you will be using an absolute file path location.

CFAppender

Property	Type	Required	Default	Description
logType	string(file or application)	false	file	The type of cflog to use: file or application.
fileName	string	false	Appender's name	The name of the file to log to if using file as the logType. If not set, it will use the appender's name

This appender logs directly to the *cflog* tag by using a custom file or logging to the application logs.

ColdBoxTracerAppender

Property	Type	Required	Default	Description
coldbox_app_key	string	false	---	The app key where ColdBox is stored in application scope because this appender talks to ColdBox via the ColdBox Factory. Don't set it to use the default

This appender logs to the ColdBox Tracer Messages Panel in the ColdBox debugger.

DBAppender & AsyncDBAppender

Property	Type	Required	Default	Description
dsn	string	true	---	The dsn to use for logging
table	string	true	---	The table name to use for logging
columnMap	struct	false	---	A column map for aliasing columns. (Optional)
autocreate	boolean	false	false	if true, then we will create the table. Defaults to false (Optional)
textDBType	string	false	*text*	The database type for the message and extended info fields.

The columns needed or created in the table are

- id : UUID
- severity : string
- category : string
- logdate : timestamp
- appendername : string
- message : string
- extrainfo : string

If you are building a column mapper, the map must have the above keys in it that match to your own table columns.

Important: Please make sure you update the *textDBType* property to match your database capabilities for logging.

EmailAppender

Property	Type	Required	Default	Description
subject	string	true	---	Get's pre-pended with the severity and category field.
from	string	true	---	The from email address
to	string	true	---	The to email(s)
cc	string	false	empty	The cc email(s)
bcc	string	false	empty	The bcc email(s)
mailserver	string	false	empty	The optional mail server
mailusername	string	false	empty	The optional mail username
mailpassword	string	false	empty	The optional mail password
mailport	int	false	25	The optional mail port
useTLS	boolean	false	false	Use the Transport level security setting in the cfmail tag.

useSSL	boolean	false	false	Use SSL or not

ScopeAppender

Property	Type	Required	Default	Description
scope	string	false	request	The scope to persist to, any valid CF scope.
key	string	false	appender's name	The key to use in the scope
limit	numeric	false	0	a limit to the amount of logs to rotate. Defaults to 0, unlimited (optional)

SocketAppender

Property	Type	Required	Default	Description
host	string	true		The host to connect to
port	string	true	---	The port to connect to
timeout	numeric	false	5	the timeout in seconds. defaults to 5 seconds
persistConnection	boolean	false	true	Whether to persist the connection or create a new one every log time. Defaults to true

TracerAppender

This appender directs messages via the *cftrace* tag. It has no configuration properties.

TwitterAppender

Property	Type	Required	Default	Description
username	string	true		The twitter username that will send the messages
password	string	true	---	The twitter password that will send the messages
logType	string	false	dm	Either *status* or *dm*, To either update status or a direct message. Defaults to direct message
dmuser	string	true (if logType=dm)	---	The user to send the direct message to

LogBox in a ColdBox Application

Every ColdBox application can use LogBox by default since the main engine already uses it. By default ANY ColdBox application will be configured with a LogBox instance with the following appenders:

- ConsoleAppender

```
<LogBox>

 <-- <Default Appender Definitions -->
 <Appender name="ConsoleAppender" class="coldbox.system.logging.appenders.ConsoleAppender" />

 <-- <Root Logger: Will log anything by default -->
 <Root levelMin="FATAL" levelMax="INFO" appenders="*" />

</LogBox>
```

Also, the app will log on ANY severity by default up to *INFO* for the root logger and the ColdBox package. You can customize this default behavior by creating or modifying the <*LogBox*> element in your ColdBox configuration file and follow the same configuration approach as any normal LogBox configuration file; please refer to the configuring LogBox section. In ColdBox 3.0.0 applications and above you can either use the XML or configuration DSL approach.

Configuration Within ColdBox

There are several ways you can configure LogBox from within your ColdBox applications, to each its own. So we will start with the two ways you can configure a ColdBox application.

Programmatic

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

- Is there a *logBox* variable defined in the configuration?
  - False:
    - Does a **LogBox.cfc** ***exist in the application's*** **config** ***folder?***
      - True: Use that CFC by convention to configure LogBox
      - False: Continue to next point
    - Configure LogBox with default framework settings
  - True:
    - Have you defined a *configFile* key?
      - True: Then use that value to pass into the configuration object so it can load LogBox using that configuration file (xml or CFC)
      - False: The configuration data is going to be defined inline here so process it

So 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).

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 LogBox configuration somewhere and you are pointing to it via this setting:

```
<LogBox>
 <ConfigFile>shared.path.LogBoxConfig</ConfigFile>
</LogBox>
```

Please also note that the application loader follows almost the same approach as above:

- Is there a <*LogBox*> element defined in the configuration?
  - False:
    - Does a **LogBox.cfc** ***exist in the application's*** **config** ***folder?***
      - True: Use that CFC by convention to configure LogBox
      - False: Continue to next point
    - Configure LogBox with default framework settings
  - True:
    - Have you defined a <*ConfigFile*> element?
      - True: Then use that value to pass into the configuration object so it can load LogBox using that configuration file (xml or CFC)
      - False: The configuration data is going to be defined inline via the XML notation here so process it

Benefits of using LogBox in a ColdBox application

Just by building a ColdBox application, you get several key benefits when dealing with LogBox.

- First of all, the configuration, creation and instantiation is ALL done for you.
- The ColdBox configuraiton file accepts ${setting} placeholder syntax, so you can make your LogBox configuration dynamic and use your ColdBox settings. You can even make your properties to appenders to be complex variables as you have the JSON notation available in the ColdBox application already or forget about that nonsense and use the ColdBox Programatic approach.
- You can configure LogBox on a per-environment criteria as the ColdBox per-environment routines can use the LogBox configuration elements in its definitions.
- Every handler, interceptor and plugin already has a reference to the LogBox instance as a property called: *logBox*. So you have immediate access to it.
- Every handler, interceptor and plugin already has a configured logger instance as a property called: *log*. So you have immediate access to it.
- The ColdBox Logger plugin is already configured to use the configured LogBox instance and you can inject it anywhere you like.
- You can configure the logging of ColdBox on a per package level.
- You get the power of ColdBox MVC.

Where is LogBox stored in a ColdBox app?

The LogBox instance is stored in the ColdBox main controller object and you can retrieve it like so from any handler, plugin or interceptor.

```
logBox = getController().getLogBox();
```

LogBox from the ColdBox Factory

The ColdBox factory object also has three utility methods you can use to talk to LogBox:

- **getLogBox()** : Get a reference to the LogBox instance.
- **getRootLogger()** : Get a reference to the root logger.
- **getLogger(category:any)** : Get a named logger instance or reference.

LogBox from the ColdBox Proxy

The ColdBox proxy object also has three utility methods you can use to talk to LogBox from any remote proxy you create:

- **getLogBox()** : Get a reference to the LogBox instance.
- **getRootLogger()** : Get a reference to the root logger.
- **getLogger(category:any)** : Get a named logger instance or reference.

Can I still use the Logger plugin?

Yes, of course. The Logger plugin (v 3.0.0 > only) has been reconfigured to work with LogBox. You can use it like any normal Logger plugin with some added methods to it. In summary, the Logger plugin is configured to work via one logger who's category name is the name of your application.

The LogBox autowiring DSL

The ColdBox autowiring DSL can also talk to LogBox. This way you can easily use our dependency injection DSL for LogBox related objects:

Type	Description
logbox	Get a reference to the application's LogBox instance
logbox:root	Get a reference to the root logger
logbox:logger:category	Get a reference to a named logger by its category name
logbox:logger:{this}	Get a reference to a named logger according to the current class path of the injected target

Below you can see the most common usage of this dependency DSL:

```
<---  LogBox wired in --->
<cfproperty name="logBox" type="logbox" />

<---  Root Logger --->
<cfproperty name="logger" type="logbox:root" />

<---  Named Category For an Object, will grab the category name from the object itself. --->
<cfproperty name="logger" type="logbox:logger:#getMetadata(this).name#" />

<---  Named Category --->
<cfproperty name="logger" type="logbox:logger:com.api.model" />

<---  Category eq to ClassPath --->
<cfproperty name="logger" type="logbox:logger:{this}" />
```

Summary

As you can see, LogBox is both a powerful and simple logging library for ColdFusion. You have great flexibility by being able to define more than 1 destination points and even building your own. The *logger* interface is incredibly easy to use and configure for any kind of custom severity levels or even destinations. LogBox is also incredibly friendly when dealing with messages as you can even customize them as you see fit.

Overall, LogBox is more than a simple logging library but an enterprise logging machine!