

[<< Back to Dashboard](#)

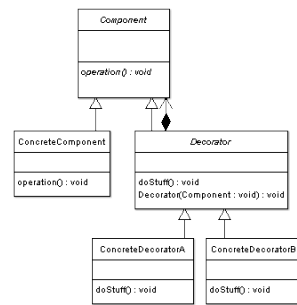
Request Context Decorator

Covers up to version 3.5.0

Introduction

The request context object is bound to the framework release and as we all know, each application is different in requirements and architecture. Thus, we have the application of the **Decorator Pattern** to our request context object in order to help developers program to their needs. So what does this mean? Plain and simply, you can decorate the ColdBox request context object with one of your own. You can extend the functionality to the specifics of the software you are building. You are not modifying the framework code base, but extending it and building on top of what ColdBox offers you.

"A decorator pattern allows new/additional behavior to be added to an existing method of an object dynamically." by [Wikipedia](#)

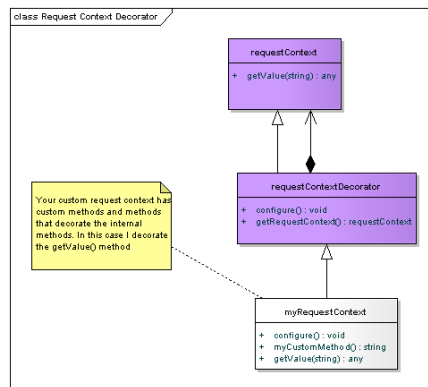


For what can I use this?

- You can use the request context decorator to add extra functionality to a request object.
- You can override the original framework event methods to meet your needs.
- These new functionalities can be specific to your application and architectural needs.
- Have different contexts for different protocols.
- and much more.

How does it work

The very first step is to create your own request context decorator component. You can see in the diagram below of the ColdBox request context design pattern.



Create a component that extends `coldbox.system.web.context.RequestContextDecorator`, this is to provide all the functionality of an original request context decorator as per the design pattern. Once you have done this, you will create a `configure` method that you can use for custom configuration when the request context gets created by the framework. Then it's up to you to add your own methods or override the original request context methods. (See [ADU](#)). In order to access the original request context object you will use the provided method called: `getRequestContext()`.

Declaration

In the sample below, you can see a custom request context declaration and a configure method. In this example, the configure method will trim all simple values that are found in the request collection.

```
<cfcomponent name="myRequestContext"
  hint="This is a simple custom request context"
  output="false"
  extends="coldbox.system.web.context.RequestContextDecorator">

<cffunction name="Configure" access="public" returnType="void" hint="This is the configuration method for your context as its created." output="false" >
  <cfset var key = "">
  <!-- Get the collection via the original request context object -->
  <cfset var rc = getRequestContext().getCollection()>
  <!-- I will trim all of the request collection -->
  <cfloop collection="#rc#" item="key">
    <!-- Trim only simple values -->
    <cffif isSimpleValue(rc[key])>
      <cfset rc[key] = trim(rc[key])>
    </cffif>
  </cfloop>
</cffunction>
</cfcomponent>
```

Sample Method Decoration

Below is a simple method decoration for the `getValue()` method. This sample basically trims any simple results as it returns them and also returns an empty variable if the value doesn't exist.

```
<cffunction name="getValue" returnType="Any" access="Public" output="false">
<cfargument name="name" type="string" required="true">
<cfargument name="defaultValue" type="any" required="false" default="NONE">
<!-- ***** -->
<cfscript>
var originalValue = "";

//Check if the value exists via original object, else return blank
if( getRequestContext().valueExists(arguments.name) ){
originalValue = getRequestContext().getValue(argumentCollection-arguments);
//check if simple
if( isSimpleValue(originalValue) ){
originalValue = trim(originalValue);
}
}
}
```

Contents

- [Request Context Decorator](#)
- [Introduction](#)
- [For what can I use this?](#)
- [How does it work](#)
 - [Declaration](#)
 - [Sample Method Decoration](#)
 - [Controller Calling](#)
- [Configuration](#)
- [Conclusion](#)

```

return originalValue;
</cfscript>
</cffunction>

```

As you can see from the code above, the possibilities to change behavior are endless. It is up to your specific requirements and it's easy!

Controller Calling

The request context decorator receives a reference to the ColdBox controller object and it can be retrieved by using the `getController()` method. This will give you the ability to call plugins, settings, interceptions and much more, right from your decorator object. This really enhances your request context object, by being able to talk to any part of your application:

```

<!-- Let's use the json plugin to inflate a value -->
<cffunction name="getValue" returnType="Any" access="Public" output="false">
<!--***** -->
<cfargument name="name" type="string" required="true">
<cfargument name="defaultValue" type="any" required="false" default="NONE">
<!--***** -->
<cfscript>
var originalValue = "";

//Check if the value exists via original object, else return blank
if( getRequestContext().valueExists(arguments.name) ){
originalValue = getRequestContext().getValue(argumentCollection=arguments);
//check if simple
if( isSimpleValue(originalValue) ){
/* json decoding*/
if ( ( left(originalValue,1) eq "[" AND right(originalValue,1) eq "]" ) OR
(left(originalValue,1) eq "{" AND right(originalValue,1) eq "}")
){
originalValue = getController().getPlugin("json").decode(replace(originalValue,"'", "''", "all"));
}
}
}

return originalValue;
</cfscript>
</cffunction>

```

Configuration

The last step in this guide is to show you how to declare your decorator in your ColdBox configuration file. Look at the sample snippet below:

```
coldbox.requestContextDecorator = "model.myRequestContext";
```

The value of the setting is the instantiation path of your request context decorator CFC.

Conclusion

As you can see from this guide, constructing ColdBox Request Context Decorators are very easy. You have a pre-defined structure that you need to implement and an easy setting to configure. You now have a great way to expand on your request requirements and add additional functionality without modifying the framework. The possibilities are endless with this feature. Enjoy!