

[← Back to Dashboard](#)

URL Mappings (SES Support)

Covers up to version 3.5.0

Introduction

ColdBox URL Mappings will give you support for creating Search Engine Safe (SES) URLs, RESTful services, and overall URL routing **your way**. We would love to take credit for this feature, but it was inspired by Rails and our good friend Adam Fortuna's ColdCourse project. By convention URL mapping support will allow you to create URL's without using the *event=this.thats¶m=val* formats by more like the following example:

```
// Old Style
http://localhost/index.cfm?event=home.about&page=2
http://localhost/index.cfm?city=24&page=3&county=234324324
```

You can have the same event but with a URL like this:

```
// Routing Style
http://localhost/home/about/page/2
http://localhost/dade/miami/page/3
```

What is a route?

A route is a declared URL pattern that if matched it will translate such URL into either an event or a view to be dispatched.

Benefits

There are several benefits that you will get by using our routing system:

- Complete control of how URL's are built and maintained
- Ability to create or build URL's dynamically
- Technology hiding
- Greater application portability
- URL's are more descriptive and easier to remember

Requirements

By default all ColdBox application templates and generated applications have SES support built in via our **SES** interceptor. This is declared in your configuration file [ColdBox.cfc](#). This will allow you to build URI's by forwarding them through the `index.cfm`.

Important: Some J2EE servlet containers do not support the forwarding of SES parameters via the routing template out of the box. You might need to enable full URL rewriting either through a web server or a J2EE filter.

```
http://localhost/index.cfm/home/about
```

However, if you would like to see minimal URI's (those without `index.cfm` in the URL) like:

```
http://localhost/home/about
```

Then you will need to enable URL rewriting at the web server level or use a J2EE rewrite filter. The most common are listed below:

Some Resources

- [Apache mod_rewrite](#) via *htaccess* or configuration files (Free)
- [Helicon Tech](#) ISAPI rewrite filter for IIS (Paid)
- [IIS7](#) native rewrite filter (Free)
- [maxim](#) native web server (free)
- [Tuckey](#) J2EE rewrite filter (free)
- [ColdBox - Tomcat Rewriting using Tuckey](#)

Our ColdBox bundle download includes an SES folder that provides you with all the necessary rewrite rules for all the resources mentioned.

Rewrite Rules

Here are just a few of those rewrite rules for you:

```
.htaccess

RewriteEngine on

# Bypass call related to administrators or non rewrite folders, you can add more here.
RewriteCond %{REQUEST_URI} !/(.*(CFIDE|cfide|CFFormGateway|jrunscripts|railo-context|mapping-tag|fckeditor)).*$
RewriteRule ^(.*)$ - [NC,L]

# Bypass flash / flex communication
RewriteCond %{REQUEST_URI} !/(.*(flashservices|flex2gateway|flex-remoting)).*$
RewriteRule ^(.*)$ - [NC,L]

# Bypass images, css, javascript and docs, add your own extensions if needed.
RewriteCond %{REQUEST_URI} !\.(bmp|gif|jpe?g|png|css|js|txt|pdf|doc|xls|ico)$
RewriteRule ^(.*)$ - [NC,L]

# The ColdBox index.cfm/{path_info} rules.
RewriteRule ^$ index.cfm [QSA,NS]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ index.cfm/${REQUEST_URI} [QSA,L,NS]
```

IsapiRewrite.ini

```
# Helicon ISAPI_Rewrite configuration file
# Version 3.1.0.48
RewriteEngine On
RepeatLimit 0

#dealing with cf-administrator,railo-administrator, etc
RewriteRule ^/(CFIDE|cfide|CFFormGateway|jrunscripts|railo-context|fckeditor) - [L,I]

#dealing with flash / flex communication, cf-administrator,railo-administrator, etc
RewriteRule ^/(flashservices|flex2gateway|flex-remoting) - [L,I]

RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ index.cfm/${REQUEST_URI} [QSA,L]
```

IIS7 web.config

```
<configuration>
  <systemWebServer>
    <rewrite>
      <rules>
        <rule name="Application_Administrat&#220;Processing&#224;ue" >
          <matchurl*"(.*)$?/*>
          <condition logicalGrouping="matchAll" >
            <add input="{SCRIPT_NAME}pattern"/!(.*(CFIDE|cfide|CFFormGateway|jrunscripts|railo-context|fckeditor|nonCaseFalse?/*>
          </condition>
          <action type="None"/>
        </rule>
        <rule name="Flash and Flex Communicat&#220;Processing&#224;ue" >
          <matchurl*"(.*)$?/*ignoreCase=false?/*>
          <condition logicalGrouping="matchAll" >
            <add input="{SCRIPT_NAME}pattern"/!(.*(flashservices|flex2gateway|flex-remoting|ignoreCaseFalse?/*>
          </condition>
          <action type="Rewrite" url="index.cfm/{PATH_INFO}appendQueryString?/*>
        </rule>
        <rule name="Static Files&#220;Processing&#224;ue" >
          <matchurl*"(.*)$?/*>
          <condition logicalGrouping="matchAll" >
            <add input="{SCRIPT_NAME}pattern"/!(.*(bmp|gif|jpe?g|png|css|js|txt|pdf|doc|ignoreCaseFalse?/*>
          </condition>
          <action type="None"/>
        </rule>
        <rule name="Insert index.cfm&#220;Processing&#224;ue" >
          <matchurl*"(.*)$?/*ignoreCase=false?/*>
          <condition logicalGrouping="matchAll" >
            <add input="{REQUEST_FILENAME}matchType=File?negate=true?/*>
            <add input="{REQUEST_FILENAME}matchType=Directory?negate=true?/*>
          </condition>
          <action type="Rewrite" url="index.cfm/{PATH_INFO}appendQueryString?/*>
        </rule>
      </rules>
    </rewrite>
  </systemWebServer>
</configuration>
```

SES Interceptor

The **SES** interceptor is the class in ColdBox that provides you with URL Mapping and RESTful support. You will have this declared (or need to declare) in your [Configuration/CFC](#):

```
interceptors = {
  [class=coldbox.system.interceptor.q.SES*
];
```

By convention the interceptor will look for `config/Routes.cfm` as your configuration file. If you want to change this, then declare a property of the interceptor with a path to your configuration file:

```
interceptors = {
  [class=coldbox.system.interceptor.q.SES*
  properties = {configFile=conf/path/Routes.cfm}
];
```

Once the SES interceptor loads in your application it will create two settings for you:

- **SESBaseURL**: The location path to your application that will be setup in your `Routes.cfm`

Contents

- [URL Mappings \(SES Support\)](#)
 - [Introduction](#)
 - [What is a route?](#)
 - [Benefits](#)
 - [Requirements](#)
 - [Some Resources](#)
 - [Rewrite Rules](#)
 - [IsapiRewrite.ini](#)
 - [IIS7 web.config](#)
- [Routes Configuration](#)
- [Configuration Methods](#)
- [Routes.cfm](#)
- [URL Mappings](#)
 - [Routing Rules](#)
 - [Routing Convention](#)
 - [Handler Routing](#)
 - [RESTful System Routing](#)
 - [View Routing](#)
 - [Pattern Placeholders](#)
 - [Syntax Placeholders](#)
 - [Alpha Placeholders](#)
 - [Dynamic handler action Placeholders](#)
 - [Routing Expression Placeholder](#)
 - [constants](#)
 - [constraints](#)
 - [Default Placeholders](#)
 - [Adding variables per route](#)
 - [URL Mapping Namespace](#)
 - [Routing Rules](#)
 - [Web Resources](#)
 - [PathInfo Provides](#)
 - [Action Build Links](#)
 - [HTML5 Base Tag](#)

• **HTMLBaseURL**: The same path as SESBaseURL but without any *index.cfm* in it (Just in case you are using *index.cfm* rewrite). This is a setting used most likely by the HTML `<base>` tag.

Routes Configuration

Inside of your *Routes.cfm* template is where you will use our routing DSL (Domain Specific Language) to define configuration parameters for your routing, RESTful URIs and to create URL mappings.

Important: The routes configuration file gets executed within the interceptor, so ALL interceptor methods are available for your usage. You can use tier detection, settings, etc.

Configuration Methods

You can use the following methods to fine tune the configuration and operation:

Method	Description
<code>setEnabled(boolean)</code>	Enable/Disable routing, enabled by default
<code>setUniqueURLs(boolean)</code>	Enables SES only URL's with permanent redirects for non-ses urls. Default is true, but we highly encourage you to always use false to allow for dual types of URLs. If true and a URL is detected with <i>?or</i> & then the application will do a 301 Permanent Redirect and try to translate the URL to a valid SES URL.
<code>setBaseURL(string)</code>	The base URL to use for URL writing and relocations. This is usually includes the web address to your application whether it is in the root or embedded in a folder. e.g. http://www.coldbox.org/ http://mysite.com/index.cfm
<code>setLooseMatching(boolean)</code>	By default URL pattern matching starts at the beginning of the URL, however, you can choose loose matching so it searches anywhere in the URL.
<code>setExtensionDetection(boolean)</code>	By default ColdBox detects URL extensions like <i>json, xml, html, pdf</i> which can allow you to build awesome RESTful web services.
<code>setValidExtensions(list)</code>	Tell the interceptor what valid extensions your application can listen to. By default it listens to: json, json!, xml, cfm, cfml, html, rss, pdf
<code>setThrowOnInvalidExtensions(boolean)</code>	By default ColdBox does not throw an exception when an invalid extension is detected. If true, then the interceptor will throw a 406 Invalid Requested Format Extension: (extension) exception.
<code>setAutoReload(boolean)</code>	By default all URL mapping routes are processed on application startup and cached. You can enable auto reloading of the routes in each request just in case you are in development and want to see change of URL mappings immediately. TURN TO FALSE IN PRODUCTION
<code>includeRoutes(path)</code>	Gives you the ability to include other routing configuration files. Great for separation and module SES rewriting.

```
setUniqueURLs false;
//Auto reload configuration, true in dev makes sense to reload the routes on every request
setAutoReload true;
// Sets automatic route extension detection and places the extension in the rc.format variable
setExtensionDetection true;
// The valid extensions this interceptor will detect
setValidExtensions( json, rss, pdf, html, xml );
// If enabled, the interceptor will throw a 406 exception that an invalid format was detected or just ignore it
setThrowOnInvalidExtensions false;

// Base URL
if len(getSetting('AppMapping')) <= 1 {
    setBaseURL( http://#cgi.HTTP_HOST#/index.cfm );
}
else {
    setBaseURL( http://#cgi.HTTP_HOST/#getSetting('AppMapping')#/index.cfm );
}

// Include some additional routes
includeRoutes( 'config/package1' );
includeRoutes( 'config/admin' );
```

All SES methods can be concatenated with each other

Base URL

The base URL is a very important configuration as that value is used when relocating and building URLs in the application. So if you are not using a rewrite engine then you will need to add the *index.cfm* to the setting. If you are using a rewrite engine, then just leave the domain entry point to the application. If you will be using the same application in a multi-tenant application, meaning one application via multiple domains, then one SES Base URL doesn't really make sense as the first domain request takes precedence. For this, you will need to modify the SES Base URL setting on a request basis. We recommend building an interceptor for this, we have one already for you in your awesome code repository [InterceptBaseURL](#). The interceptor basically tells the request context what URL to use for that request.

```
component {
    function process( event, interceptData ) {
        event.setSESBaseURL( http://#cgi.HTTP_HOST );
    }
}
```

URL Mappings

URL mapping or routing is done via our routing methods:

- `addRoute()` - Add URL routing to controllers and actions with RESTful capabilities
- `addNamespace()` - Ability to group routes on a specific URL pattern entry point
- `addModuleRoutes()` - Ability to register a specific URL pattern entry point for a [ColdBox Module](#)
- `with()`, `endWith()` - Context methods that will allow you to prefix or default repetitive patterns in the arguments of `addRoute()`

Adding Routes

```
public any addRoute( string pattern, [ string handler ], [ any action ], [ boolean packageResolverExempt='false' ], [ string matchVariables ], [ string view ], [ boolean viewNoLayout='false' ], [ boolean valuePairTranslation='true' ], [ any
```

Argument	Type	Required	Default	Description
pattern	string	true	---	The URL pattern to look for in the incoming URL
handler	string	false	---	The handler to translate to, can include module or package path
action	string or struct	false	---	The action to translate to. If a structure, then the keys represent the HTTP verbs of the RESTful action to relocate to.
packageResolverExempt	boolean	false	true	Automatically can resolve packages in the URL when using routing by convention
matchVariables	string	false	---	A query string of variables to inject into the request collection if the route matches
view	string	false	---	The name of the view to dispatch the URL to instead of event routing
viewNoLayout	boolean	false	false	Use no layout when rendering the view dispatched or not
valuePairTranslation	boolean	false	true	By default it converts any name-value pair after the matched URL pattern to the request collection
constraints	struct	false	{}	A structure of regex constraints for variable placeholders in the URL patterns
module	string	false	---	Add this route to a named module
moduleRouting	boolean	false	false	Called internally by <code>addModuleRoutes</code> to add a module routing route only.
namespace	string	false	---	The namespace to add this route to
ssl	boolean	false	false	Makes the route SSL only if true, else for either SSL or non SSL. If SSL, the interceptor will relocate to the same route but in SSL.
append	boolean	false	true	By default all routes are stored in first come first served order, but you can pre-pend to the first position if so desired.

This is the meat and potatoes for enabling routing. You use this method to declare routes that will be dispatched by the interceptor. The syntax of these are similar to Ruby on Rails. The idea is that the number of variables in a URL will be the first indicator of which course to use. Basically, the interceptor goes through each rule in a top-down format and tries to match the incoming URL to the route. If a route matches it will try to create the appropriate event and extra variables in order to respond to a request.

Note: You can pass any named argument and value to this method and the interceptor will create a new variable with the name of the argument in the request collection for you. This can be used as an alternative to using the `matchVariables` argument.

Routing By Convention

The default route is:

```
addRoute( pattern=handler/:action? );
```

The URL pattern in the default route includes two special position holders:

- **handler** - The handler to relocate to (Including Package and Module)
- **action** - The action to relocate to

With one route you can potentially write an entire application because once a route is detected, any extra name-value pair in the URL will be translated to variables in the request collection for you.

```
http://localhost/general -> event=general.index
http://localhost/general/index -> event=general.index
http://localhost/general/index/id/2 -> event=general.index&id=2
// If 'admin' is a package in the handlers directory
http://localhost/admin/general/index -> event=admin.general.index
// If 'admin' is a module
http://localhost/admin/general/index/id/4/page/2 -> event=admin.general.index&id=4&page=2
```

With one route you write all your URLs. However, the problem is that your routing URLs depend too much in the name of your handlers and actions. Thus, if you refactor, your URLs change and that's not very nice. That's why we encourage you to create more routes so this can be avoided especially on public sites.

Handler Routing

Once you declare a route you will most likely route it to an event by using the `handler` and/or `action` arguments:

```
addRoute( pattern=blog; handler=blog; action=index );
```

This create the `event=blog.index` translation for you.

RESTful Action Routing

You can also use a structure for the `action` argument so you can split the incoming HTTP method verbs into the appropriate actions:

```
addRoute( pattern=user/:username; handler=user; action={
    GET => list; POST => create; PUT => save; DELETE => remove; HEAD => info
});
```

Isn't that cool! Just like that you can split the incoming URL pattern to appropriate action executions according to the incoming HTTP verb.

View Routing

You can also route a URL pattern to a view with its default or assigned layout or none at all:

```
addRoute(pattern=contact-us^view=static/contact^
addRoute(pattern=newsletter^view=static/newsletter^layout=trug;
```

Pattern Placeholders

In your URL pattern you can also use the same `:syntax` to denote a variable position holder. These position holders are alpha-numeric by default:

```
addRoute(pattern="blog/:year/:month/:day?");
```

Once a URL is matched to the route, those placeholders will become request collection variables:

```
http://localhost/blog/2012/12/22 -> rc.year=2012, rc.month=12, rc.day=22
```

Numeric Placeholder

ColdBox gives you also the ability to declare numeric only routes by appending `-numeric` to the variable placeholder so the route will only match if the placeholder is numeric.

```
/blog/:year-numeric/:month-numeric/:day-numeric?
```

That's it. Everytime you append the `-numeric` tag, the placeholder will only match if its numeric.

Alpha Placeholder

ColdBox gives you also the ability to declare alpha only routes by appending `-alpha` to the variable placeholder so the route will only match if the placeholder is alpha only.

```
/wiki/:page-alpha
```

Dynamic handler/action Placeholders

You can also use the reserved position placeholders `:handler` and `:action` so you can specifically position them in the URL dynamically, meaning their names comes from the URL:

```
// route with regex constraint
addRoute(pattern=admin/users/:action^handler=admin.user#?
addRoute(pattern=admin/:handler/:action#?)
```

Regular Expression Placeholder

There are two ways to place a regex constraint on a placeholder, using the `regex:` placeholder or adding a `constraints` structure to the route declaration.

regex:()

```
// route with regex constraint
addRoute(pattern=api/regex:(^([xml|json]))/*
    handler=api;
    action=execute;
```

Error parsing plugin definition: The MESSAGE parameter to the renderit function is required but was not passed in.

constraints

The key in the structure must match the name of the placeholder and the value is a regex expression that **must** be enclosed by parenthesis (`()`).

```
// route with custom constraints
addRoute(pattern=api/:format/*
    handler=api;
    action=execute;
    constraints={
        format: "([xml|json])*"
    });
```

Optional Placeholders

Most of the time we need to create several routes in order to determine possible routings and they must be declared in a specific order so they match. A great example is the declaration of the following:

```
/blog/:year-numeric/:month-numeric/:day-numeric
/blog/:year-numeric/:month-numeric
/blog/:year-numeric/
/blog/
```

However, we just wrote 4 routes for this when we can just use optional variables by using the `?` symbol at the end of the placeholder. This tells the processor to create the routes for you in the most detailed manner first:

```
/blog/:year-numeric?/:month-numeric?/:day-numeric?
```

Just remember that an optional placeholder cannot be followed by a non-optional one. It doesn't make sense.

Adding variables per route

You can add variables to the request collection by using the `matchVariables` argument or by adding your extra name-value pairs to the method as arguments. The `matchVariables` argument is a list of name-value pairs that you can pass to the route definition. This basically tells the processor that if that specific route matches, then add those name-value pairs to the request collection.

```
addRoute(pattern=space/:space^handler=page^action=show^matchVariables=spaceUsed=true,useNavigation=false^
```

As mentioned above, you can also add variables by just adding them as arguments to the `addRoute()` method:

```
addRoute(pattern=space/:space^handler=page^action=show^spaceUsed=true,useNavigation=false;
```

URL Mapping Namespaces

You can create a la-carte namespaces for URL routes. Namespaces are cool groupings of routes according to a specific URL entry point. So you can say that all URLs that start with `/testing` will be found in the `testing` namespace and it will iterate through the namespace routes until it matches one of them. Much how modules work, where you have a module entry point, now you can create virtual entry point to ANY route by namespaces it. This route can be a module a non-module, package, or whatever you like. You start off by registering the namespace using the `addNamespace(pattern, namespace)` method:

```
addNamespace(pattern=testing^namespace=test;
addNamespace(pattern=news^namespace=blog);
```

Once a namespace is registered you can add routes to it via the `addRoute()` method or the `with()` closure.

```
// Via addRoute
addNamespace(pattern=news^namespace=blog)
    .addRoute(pattern=handler=blog^action=index^namespace=blog)
    .addRoute(pattern=year-numeric/:month-numeric/:day-numeric^handler=blog^action=index^namespace=blog);
```

```
// Via with closure
addNamespace(pattern=news^namespace=blog);
with(namespace=blog, handler=blog)
    .addRoute(pattern=action=index)
    .addRoute(pattern=year-numeric/:month-numeric/:day-numeric^action=index);
.endWith();
```

You can also register multiple URL patterns that point to the same namespace

Module Routes

You can use the `addModuleRoutes()` to define a la-carte module entry points. By convention, all module's declared entry point in their configuration file is registered for you. However, you can use this manual method approach to create aliases or funky stuff:

```
addModuleRoutes(pattern=blog; module=blog);
addModuleRoutes(pattern=news; module=blog);
```

With Closures

We have created some cool context methods to allow for the prefixing of any of the `addRoute()` arguments by using what we call `with closures`. This allows you to prefix repetitive patterns in route declarations. The best way to see how it works is by example:

```
addRoute(pattern=news; handler=public.news^action=index);
addRoute(pattern=news/recent; handler=public.news^action=recent);
addRoute(pattern=news/remove; handler=public.news^action=remove);
addRoute(pattern=news/add/:title^handler=public.news^action=add);
addRoute(pattern=news/delete/:slug^handler=public.news^action=remove);
addRoute(pattern=news/:slug^handler=public.news^action=view);
```

As you can see from the routes above, we have lots of repetitive code that we can clean out. So let's look at the same routes but using some nice with closures.

```
with(pattern=news; handler=public.news^
    .addRoute(pattern= action=index)
    .addRoute(pattern=recent^ action=recent)
    .addRoute(pattern=remove^ action=remove)
    .addRoute(pattern=add/:title^ action=add)
    .addRoute(pattern=delete/:slug^ action=remove)
    .addRoute(pattern=:slug^ action=view)
.endWith();
```

As you can see, we start our URL mapping DSL with the `with()` method and pass in any argument the `addRoute()` method declares. In this case we pass a `pattern` and a `handler`. Meaning any `addRoute()` method that is concatenated in the with closure will be prefixed with that `pattern` and `handler`. Once we concatenate the last `addRoute()`, then we finalize the closure with the `.endWith()`; demarcation. BOOM! The patterns look so much manageable and declarable. The arguments you can use for prefixing or defaulting are:

Argument	Description
<code>pattern</code>	The URL pattern prefix
<code>handler</code>	The handler prefix
<code>action</code>	The action prefix
<code>matchVariables</code>	The match variables prefix
<code>view</code>	The view prefix
<code>constraints</code>	The constraints to prefix

module The module prefix
namespace The namespace prefix

Important: It is extremely important that you close the with closures with an `endWith()` call or all subsequent `addRoute()` calls, will be using the last with closure you declared.

PathInfo Providers

By default, the URL mapping processor will detect routes by looking at the `CGL.PATH_INFO` variable. This feature can be useful to set flags for each request based on a URL and then clean or parse the URL to a more generic form to allow for simple route declarations. Users may include internationalization (i18n) and supporting multiple experiences based on devices such as Desktop, Tablet, Mobile and TV. To modify the URI used by the SES interceptor before route detection occurs simple follow the convention of adding a UDF called `PathInfoProvider()` to your routes configuration file (`config/Routes.cfm`).

The `PathInfoProvider()` UDF is responsible for returning the string used to match a route. Without this UDF the SES interceptor will simply obtain the URI from the `CGL.PATH_INFO` variable.

```
// Example PathInfoProvider for detecting a mobile request
function PathInfoProvider(Event){
    var rc = Event.getCollection();
    var prc = Event.getCollection('private');

    local.URI = CGI.PATH_INFO;

    if (reFindNoCase('/m',local.URI) == 0)
    {
        // Does not look like this could be a mobile request...
        return local.URI;
    }

    // Mobile Request? Let's find out.

    // If the URI is '/m' it is easy to determine that this is a
    // request for the Mobile Homepage.
    if (len(local.URI) == 2)
    {
        prc.mobile=true
        // Simply return '/' since they want the mobile homepage
        return '/';
    }

    // Only continue with our mobile evaluation if we have a slash after
    // our '/m'. Without a slash following the /m the route is something
    // else like coldbox.org/makes/cool/stuff
    if (reFindNoCase('/m/;',local.URI) == 1)
    {
        // Looks like we are mobile!
        prc.mobile=true

        // Remove our '/m/' determination and continue
        // processing for languages...
        local.URI = REReplaceNoCase(local.URI, '/');
    }

    // The URI starts with an 'm' but does not look like
    // a mobile request. So, simply return the URI for normal
    // route detection...
    return local.URI;
}
```

event.buildLink()

```
public any buildLink(string linkto, [boolean translate='true'], [boolean ssl='false'], [string baseUrl=''], [string queryString=''])
```

The request context has a method called `buildLink()` that will build SES URLs for you. Just pass in the route or event and it will create the appropriate URL for you:

```
<a href=#event.buildLink('home.about')>
<a href=#event.buildLink('user.edit.id.user.getEdit')>
```

The `buildLink` will translate any periods (.) to (/) slashes for you automatically

HTML base tag

Well, for the base tag you can use either the `sesBaseUrl` setting, if using minimal URL support, or the `htmlBaseUrl` setting if using the `index.cfm` in the URL. Both of these settings are set by the interceptor at configuration time and available for your usage via the `getSetting()` method.

```
<basehref=#getSetting('htmlBaseUrl')#>
<basehref=#getSetting('sesBaseUrl')#>
```

This tells the browsers how to find your assets when using URL mappings or SES URLs.