Tutorial

The goal of this tutorial is to quickly get you started with using **Caelyf** to write and deploy Groovy applications on Cloud Foundry. We'll assume you have already setup your computer with the VMC tooling for deploying your applications to Cloud Foundry. If you haven't, please do so by reading the instructions from the Cloud Foundry site.

The easiest way to get setup quickly is to download the template project from the download section. It provides a ready-to-go project with the right configuration files pre-filled and an appropriate directory layout:

- **`web.xml`** preconfigured with the **Caelyf** servlets
- a sample Groovlet and template
- the needed JARs (Groovy, Caelyf, Jedis)

You can browse the JavaDoc of the classes composing **Caelyf**.

Table of Content

- Setting up your project
  - Directory layout
  - Configuration files
- The template project
  - Gradle build file
- Views and controllers
  - Variables available in the binding
    - Eager variables
    - Lazy variables
    - Injecting services and variables in your classes
  - Templates
    - Includes
    - Redirect and forward
  - Groovlets
    - Using MarkupBuilder to render XML or HTML snippets
    - Delegating to a view template
  - Logging messages
- Flexible URL routing system
  - Configuring URL routing
  - Defining URL routes
  - Using wildcards
  - Using path variables
    - Validating path variables
  - Ignoring certain routes
  - Caching groovlet and template output
- Simple plugin system
  - What a plugin can do for you
  - Anatomy of a plugin
    - Hierarchy
    - The plugin descriptor
  - Using a plugin
  - How to distribute and deploy a plugin

Setting up your project

Directory layout

We'll follow the directory layout proposed by the **Caelyf** template project:

```
/
+-- src
|    |
|    +-- main
|    |    |
|    |    +-- groovy
|    |    +-- java
|    |
|    +-- test
|         |
|         +-- groovy
|         +-- java
|
+-- war
     |
     +-- index.gtpl
     +-- css
     +-- images
     +-- js
     +-- WEB-INF
          |
          +-- appengine-web.xml
          +-- web.xml
          +-- plugins.groovy      // if you use plugins
          +-- routes.groovy       // if you use the URL routing system
          +-- classes
          |
          +-- groovy
          |    |
          |    +-- controller.groovy
          |
          +-- pages
          |    |
          |    +-- view.gtpl
          |
          +-- includes
          |    |
          |    +-- footer.gtpl
          |
          +-- lib
               |
               +-- caelyf-x.y.z.jar
               +-- groovy-all-x.y.z.jar
               +-- jedis-x.y.z.jar
```

At the root of your project, you'll find:

- **`src`**: If your project needs source files beyond the templates and groovlets, you can place both your Java and Groovy sources in that directory. Before running the local app engine dev server or before deploying your application to app engine, you should make sure to pre-compile your Groovy and Java classes so they are available in **`WEB-INF/classes`**.
- **`war`**: This directory will be what's going to be deployed on app engine. It contains your groovlets, templates, images, JavaScript files, stylesheets, and more. It also contains the classical **`WEB-INF`** directory from typical Java web applications.

In the **`WEB-INF`** directory, you'll find:

- **`web.xml`**: The usual Java EE configuration file for web applications.
- **`classes`**: The compiled classes (compiled with **`build.groovy`**) will go in that directory.

- **`groovy`**: In that folder, you'll put your controller and service files written in Groovy in the form of Groovlets.
- **`pages`**: Here you can put all your template views, to which you'll point at from the URL routes configuration.
- **`includes`**: We propose to let you put included templates in that directory.
- **`lib`**: All the needed libraries will be put here, the Groovy, **Caelyf** and Jedis (the Java library to access Redis, for page caching purposes), as well as any third-party JARs you may need in your application.

> ***Note:*** *You may decide to put the Groovy scripts and includes elsewhere, but the other files and directories can't be changed, as they are files the servlet container expects to find at that specific location.*

Configuration files

With the directory layout ready, let's have a closer look at the **`web.xml`** file.

web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5">
    <!-- A servlet context listener to initialize the plugin system -
       ->
    <listener>
        <listener-
           class>groovyx.caelyf.ServletContextListener</listener-
           class>
    </listener>

    <!-- The Caelyf Groovlet servlet -->
    <servlet>
        <servlet-name>GroovletServlet</servlet-name>
        <servlet-class>groovyx.caelyf.CaelyfServlet</servlet-class>
    </servlet>

    <!-- The Caelyf template servlet -->
    <servlet>
        <servlet-name>TemplateServlet</servlet-name>
        <servlet-
           class>groovyx.caelyf.CaelyfTemplateServletlt;/servlet-
           class>
    </servlet>

    <!-- The URL routing filter -->
    <filter>
        <filter-name>RoutesFilter</filter-name>
        <filter-class>groovyx.caelyf.routes.RoutesFilter</filter-
           class>
    </filter>

    <!-- Specify a mapping between *.groovy URLs and Groovlets -->
    <servlet-mapping>
        <servlet-name>GroovletServlet</servlet-name>
        <url-pattern>*.groovy</url-pattern>
    </servlet-mapping>

    <!-- Specify a mapping between *.gtpl URLs and templates -->
    <servlet-mapping>
        <servlet-name>TemplateServlet</servlet-name>
        <url-pattern>*.gtpl</url-pattern>
    </servlet-mapping>

    <filter-mapping>
        <filter-name>RoutesFilter</filter-name>
```

```
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- Define index.gtpl as a welcome file -->
    <welcome-file-list>
        <welcome-file>index.gtpl</welcome-file>
    </welcome-file-list>
</web-app>
```

In **`web.xml`**, we first define a servlet context listener, to initialize the plugin system. We define the two Caelyf servlets for Groovlets and templates, as well as their respective mappings to URLs ending with **`.groovy`** and **`.gtpl`**. We setup a servlet filter for the URL routing to have nice and friendly URLs. We then define a welcome file for **`index.gtpl`**, so that URLs looking like a directory search for and template with that default name.

> ***Note**: You can update the filter definition as shown below, when you attempt to forward to a route from another Groovlet to keep your request attributes. Without the dispatcher directives below the container is issuing a 302 redirect which will cause you to lose all of your request attributes.*
>
> ```
> <filter-mapping>
>     <filter-name>RoutesFilter</filter-name>
>     <url-pattern>/*</url-pattern>
>     <dispatcher>FORWARD</dispatcher>
>     <dispatcher>REQUEST</dispatcher>
> </filter-mapping>
> ```

Using the template project

You can use the template project offered by **Caelyf**. It uses **Gradle** for the build, and for running and deploying applications, and **Spock** for testing your groovlets. Please have a look at the template project section to know more about it.

Views and controllers

Now that our project is all setup, it's time to dive into groovlets and templates.

> ***Tip:*** *A good practice is to separate your views from your logic (following the usual MVC pattern). Since* ***Caelyf*** *provides both view templates and Groovlet controllers, it's advised to use the former for the view and the later for the logic.*

Caelyf builds on Groovy's own Groovlets and template servlet to add a shorcuts to the Cloud Foundry services.

> ***Note:*** *You can learn more about Groovy's Groovlets and templates from this article on IBM developerWorks.* ***Caelyf****'s own Groovlets and templates are just an extension of Groovy's ones, and simply decorate Groovy's Groovlets and templates by giving access to Cloud Foundry services and add some additional methods to them via Groovy categories.*

Variables available in the binding

A special servlet binding gives you direct access to some implicit variables that you can use in your views and controllers:

Eager variables

- **`request`** : the **`HttpServletRequest`** object
- **`response`** : the **`HttpServletResponse`** object
- **`context`** : the **`ServletContext`** object
- **`application`** : same as **`context`**
- **`session`** : shorthand for **`request.getSession(false)`** (can be null) which returns an **`HttpSession`**
- **`params`** : map of all form parameters (can be empty)
- **`headers`** : map of all **`request`** header fields
- **`log`** : a Groovy logger is available for logging messages through **`java.util.logging`**
- **`logger`** : a logger accessor can be used to get access to any logger (more on logging)

Lazy variables

- **`out`** : shorthand for **`response.getWriter()`** which returns a **`PrintWriter`**
- **`sout`** : shorthand for **`response.getOutputStream()`** which returns a **`ServletOutputStream`**
- **`html`** : shorthand for **`new MarkupBuilder(response.getWriter())`** which returns a **`MarkupBuilder`**

> ***Note:*** *The eager variables are pre-populated in the binding of your Groovlets and templates. The lazy variables are instantiated and inserted in the binding only upon the first request.*

Beyond those standard Servlet variables provided by Groovy's servlet binding, **Caelyf** also adds ones of his own by injecting the following:

- **`localMode`** : a boolean variable which is **`true`** when the application is running in local development mode, and **`false`** when deployed on Cloud Foundry.
- **`app`** : a map variable with the following keys and values:
  - **`caelyf`** : a map with the following keys and values:
    - **`version`** : the version of the **Caelyf** toolkit used (here: 0.1)

> ***Note:*** *Regarding the* ***`app`*** *variable, this means you can access those values with the*

> *following syntax in your groovlets and templates:*
>
> ```
> app.caelyf.version
> ```

Thanks to all these variables and services available, you'll be able to access the Cloud Foundry services and Servlet specific artifacts with a short and concise syntax to further streamline the code of your application.

Injecting services and variables in your classes

All the variables and services listed in the previous sections are automatically injected into the binding of Groovlets and templates, making their access transparent, as if they were implicit or global variables. But what about classes? If you want to also inject the services and variables into your classes, you can annotate them with the **`@CaelyfBindings`** annotation.

```
import groovyx.caelyf.CaelyfBindings

// annotate your class with the transformation
@CaelyfBindings
class WeblogService {
    def numberOfComments(post) {
        // you can use Redis right away
        redis.get('someKey')
    }
}
```

The annotation instructs the compiler to create properties in your class for each of the services and variables.

> ***Note:*** *Variables like* ***`request`***, ***`response`***, ***`session`***, ***`context`***, ***`params`***, ***`headers`***, ***`out`***, ***`sout`***, ***`html`*** *are not bound in your classes.*

> ***Note:*** *If your class already has a property of the same name as the variables and services injected by this AST transformation, they won't be overriden.*

Templates

Caelyf templates are very similar to JSPs or PHP: they are pages containing scriptlets of code. You can:

- put blocks of Groovy code inside **`<% /* some code */ %>`**,
- call **`print`** and **`println`** inside those scriptlets for writing to the servlet writer,
- use the **`<%= variable %>`** notation to insert a value in the output,
- or also the GString notation **`${variable}`** to insert some text or value.

Let's have a closer look at an example of what a template may look like:

```
<html>
    <body>
        <p><%
            def message = "Hello World!"
            print message %>
        </p>
        <p><%= message %></p>
        <p>${message}</p>
        <ul>
        <% 3.times { %>
            <li>${message}</li>
        <% } %>
```

```
        </ul>
    </body>
</html>
```

The resulting HTML produced by the template will look like this:

```
<html>
    <body>
        <p>Hello World!</p>
        <p>Hello World!</p>
        <p>Hello World!</p>
        <ul>
            <li>Hello World!</li>
            <li>Hello World!</li>
            <li>Hello World!</li>
        </ul>
    </body>
</html>
```

If you need to import classes, you can also define imports in a scriplet at the top of your template as the following snippet shows:

```
<% import com.foo.Bar %>
```

> ***Note:*** *Of course, you can also use Groovy's type aliasing with* ***`import com.foo.ClassWithALongName as CWALN`****. Then, later on, you can instantiate such a class with* ***`def cwaln = new CWALN()`****.*

> ***Note:*** *Also please note that import directives don't look like JSP directives (as of this writing).*

As we detailed in the previous section, you can also access the Servlet objects (request, response, session, context), as well as other services offered by Cloud Foundry, such as Redis. For instance, the following template will display some page content surrounded by a header and footer retrieved from the Redis cache:

```
<html>
    <body>
        <%= redis.get('page-header') %>
        <div>Some content</div>
        <%= redis.get('page-footer') %>
    </body>
</html>
```

Includes

Often, you'll need to reuse certain graphical elements across different pages. For instance, you always have a header, a footer, a navigation menu, etc. In that case, the include mechanism comes in handy. As advised, you may store templates in **`WEB-INF/includes`**. In your main page, you may include a template as follows:

```
<% include '/WEB-INF/includes/header.gtpl' %>

<div>My main content here.</div>

<% include '/WEB-INF/includes/footer.gtpl' %>
```

Redirect and forward

When you want to chain templates or Groovlets, you can use the Servlet redirect and forward capabilities. To do a forward, simply do:

```
<% forward 'index.gtpl' %>
```

For a redirect, you can do:

```
<% redirect 'index.gtpl' %>
```

Groovlets

In contrast to view templates, Groovlets are actually mere Groovy scripts. But they can access the output stream or the writer of the servlet, to write directly into the output. Or they can also use the markup builder to output HTML or XML content to the view.

Let's have a look at an example Groovlet:

```
println """
    <html>
        <body>
"""

[1, 2, 3, 4].each { number -> println "<p>${number}</p>" }

def now = new Date()

println """
            <p>
                ${now}
            </p>
        </body>
    </html>
"""
```

You can use **`print`** and **`println`** to output some HTML or other plain-text content to the view. Instead of writing to **`System.out`**, **`print`** and **`println`** write to the output of the servlet. For outputing HTML or XML, for instance, it's better to use a template, or to send fragments written with a Markup builder as we shall see in the next sessions. Inside those Groovy scripts, you can use all the features and syntax constructs of Groovy (lists, maps, control structures, loops, create methods, utility classes, etc.)

Using `MarkupBuilder` to render XML or HTML snippets

Groovy's **`MarkupBuilder`** is a utility class that lets you create markup content (HTML / XML) with a Groovy notation, instead of having to use ugly **`println`**s. Our previous Groovlet can be written more cleanly as follows:

```
html.html {
    body {
        [1, 2, 3, 4].each { number -> p number }

        def now = new Date()

        p now
    }
}
```

> ***Note:*** *You may want to learn more about* ***`MarkupBuilder`*** *in the Groovy wiki documentation or on this article from IBM developerWorks.*

Delegating to a view template

As we explained in the section about redirects and forwards, at the end of your Groovlet, you may simply redirect or forward to a template. This is particularly interesting if we want to properly decouple the logic from the view. To continue improving our previous Groovlets, we may, for instance, have a Groovlet compute the data needed by a template to render. We'll need a Groovlet and a template. The Groovlet **`WEB-INF/groovy/controller.groovy`** would be as follows:

```
request['list'] = [1, 2, 3, 4]
request['date'] = new Date()

forward 'display.gtpl'
```

> ***Note:*** *For accessing the request attributes, the following syntaxes are actually equivalent:*
>
> ```
> request.setAttribute('list', [1, 2, 3, 4])
> request.setAttribute 'list', [1, 2, 3, 4]
> request['list'] = [1, 2, 3, 4]
> request.list = [1, 2, 3, 4]
> ```

The Groovlet uses the request attributes as a means to transfer data to the template. The last line of the Groovlet then forwards the data back to the template view **`display.gtpl`**:

```
<html>
    <body>
    <% request.list.each { number -> %>
        <p>${number}</p>
    <% } %>
        <p>${request.date}</p>
    </body>
</html>
```

Logging messages

In your Groovlets and Templates, thanks to the **`log`** variable in the binding, you can log messages through the **`java.util.logging`** infrastructure. The **`log`** variable is an instance of **`groovyx.caelyf.logging.GroovyLogger`** and provides the methods: **`severe(String)`**, **`warning(String)`**, **`info(String)`**, **`config(String)`**, **`fine(String)`**, **`finer(String)`**, and **`finest(String)`**.

The default loggers in your groovlets and templates follow a naming convention. The groovlet loggers' name starts with the **`caelyf.groovlet`** prefix, whereas the template loggers' name starts with **`caelyf.template`**. The name also contains the internal URI of the groovlet and template but transformed: the slashes are exchanged with dots, and the extension of the file is removed.

> ***Note:*** *The extension is dropped, as one may have configured a different extension name for groovlets and templates than the usual ones (ie.* ***`.groovy`*** *and* ***`.gtpl`****).*

A few examples to illustrate this:

URI	Logger name
`/myTemplate.gtpl`	**`caelyf.template.myTemplate`**
`/crud/scaffolding.gtpl`	**`caelyf.template.crud.scaffolding`**

/WEB-INF/templates/aTemplate.gtpl	**caleyf.template.WEB-INF.templates.aTemplate**
/upload.groovy (ie. **/WEB-INF/groovy/upload.groovy**)	**caelyf.groovlet.upload**
/account/credit.groovy (ie. **/WEB-INF/groovy/account/credit.groovy**)	**caelyf.groovlet.account.credit**

This naming convention is particularly interesting as the **java.util.logging** infrastructure follows a hierarchy of loggers depending on their names, using dot delimiters, where **caelyf.template.crud.scaffolding** inherits from **caelyf.template.crud** which inherits in turn from **caelyf.template**, then from **caelyf**. You get the idea! For more information on this hierarchy aspect, please refer to the Java documentation.

Concretely, it means you'll be able to have a fine grained way of defining your loggers hierarchy and how they should be configured, as a child inherits from its parent configuration, and a child is able to override parent's configuration. So in your **logging.properties** file, you can have something like:

```
# Set default log level to INFO
.level = INFO

# Configure Caelyf's log level to WARNING, including groovlet's and template's
caelyf.level = WARNING

# Configure groovlet's log level to FINE
caelyf.groovlet.level = FINE

# Override a specific groovlet familty to FINER
caelyf.groovlet.crud.level = FINER

# Set a specific groovlet level to FINEST
caelyf.groovlet.crud.scaffoldingGroovlet.level = FINEST

# Set a specific template level to FINE
caelyf.template.crud.editView.level = FINE
```

You can also use the **GroovyLogger** in your Groovy classes:

```
import groovyx.caelyf.logging.GroovyLogger
// ...
def log = new GroovyLogger("myLogger")
log.info "This is a logging message with level INFO"
```

It is possible to access any logger thanks to the logger accessor, which is available in the binding under the name **logger**. From a Groovlet or a Template, you can do:

```
// access a logger by its name, as a property access
logger.myNamedLogger.info "logging an info message"

// when the logger has a complex name (like a package name with
   dots), prefer the subscript operator:
logger['com.foo.Bar'].info "logging an info message"
```

Additionally, there are two other loggers for tracing the routes filter and plugins handler, with **caelyf.routesfilter** and **caelyf.pluginshandler**. The last two log their messages with the **CONFIG** level, so be sure to adapt the logging level in your logging configuration file if you wish to troubleshoot how routes and plugins are handled.

Flexible URL routing

Caelyf provides a flexible and powerful URL routing system: you can use a small Groovy Domain-Specific Language for defining routes for nicer and friendlier URLs.

Configuring URL routing

To enable the URL routing system, you should configure the **`RoutesFilter`** servlet filter in **`web.xml`**:

```
...
<filter>
    <filter-name>RoutesFilter</filter-name>
    <filter-class>groovyx.caelyf.routes.RoutesFilter</filter-class>
</filter>
...
<filter-mapping>
    <filter-name>RoutesFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
...
```

> ***Note:*** *We advise to setup only one route filter, but it is certainly possible to define several ones for different areas of your site. By default, the filter is looking for the file* ***`/WEB-INF/routes.groovy`*** *for the routes definitions, but it is possible to override this setting by specifying a different route DSL file with a servlet filter configuration parameter:*
>
> ```
> <filter>
>     <filter-name>RoutesFilter</filter-name>
>     <filter-class>groovyx.caelyf.routes.RoutesFilter</filter-
>        class>
>     <init-param>
>         <param-name>routes.location</param-name>
>         <param-value>/WEB-INF/blogRoutes.groovy</param-value>
>     </init-param>
> </filter>
> ```

> ***Warning:*** *The filter is stopping the chain filter once a route is found. So you should ideally put the route filter as the last element of the chain.*

Defining URL routes

By default, once the filter is configured, URL routes are defined in **`WEB-INF/routes.groovy`**, in the form of a simple Groovy scripts, defining routes in the form of a lightweight DSL. The capabilities of the routing system are as follow, you can:

- match requests made with a certain method (GET, POST, PUT, DELETE), or all
- define the final destination of the request
- chose whether you want to forward or redirect to the destination URL (i.e. URL rewriting through forward vs. redirection)
- express variables in the route definition and reuse them as variables in the final destination of the request
- validate the variables according to some boolean expression, or regular expression matching
- cache the output of groovlets and templates pointed by that route for a specified period of time

Let's see those various capabilities in action. Imagine we want to define friendly URLs for our blog application. Let's configure a first route in **`WEB-INF/routes.groovy`**. Say you want to provide a shorthand URL **`/about`** that would redirect to your first blog post. You could configure the **`/about`** route for all GET requests calling the **`get`** method. You would then redirect those requests to the final destination with the **`redirect`** named argument:

```
get "/about", redirect: "/blog/2008/10/20/welcome-to-my-blog"
```

If you prefer to do a forward, so as to do URL rewriting to keep the nice short URL, you would just replace **`redirect`** with **`forward`** as follows:

```
get "/about", forward: "/blog/2008/10/20/welcome-to-my-blog"
```

If you have different routes for different HTTP methods, you can use the **`get`**, **`post`**, **`put`** and **`delete`** methods. If you want to catch all the requests independently of the HTTP method used, you can use the **`all`** function. Another example, if you want to post only to a URL to create a new blog article, and want to delegate the work to a **`post.groovy`** Groovlet, you would create a route like this one:

```
post "/new-article", forward: "/post.groovy"     // shortcut for
   "/WEB-INF/groovy/post.groovy"
```

> ***Note:*** *When running your applications in development mode,* ***Caelyf*** *is configured to take into accounts any changes made to the* ***`routes.groovy`*** *definition file. Each time a request is made, which goes through the route servlet filter,* ***Caelyf*** *checks whether a more recent route definition file exists. However, once deployed on Cloud Foundry, the routes are set in stone and are not reloaded. The sole cost of the routing system is the regular expression mapping to match request URIs against route patterns.*

Using wildcards

You can use a single and a double star as wildcards in your routes, similarly to the Ant globing patterns. A single star matches a word (**`/\w+/`**), where as a double start matches an arbitrary path. For instance, if you want to show information about the blog authors, you may forward all URLs starting with **`/author`** to the same Groovlet:

```
get "/author/*", forward: "/authorsInformation.groovy"
```

This route would match requests made to **`/author/johnny`** as well as to **`/author/begood`**.

In the same vein, using the double star to forward all requests starting with **`/author`** to the same Groovlet:

```
get "/author/**", forward: "/authorsInformation.groovy"
```

This route would match requests made to **`/author/johnny`**, as well as **`/author/johnny/begood`**, or even **`/author/johnny/begood/and/anyone/else`**.

> ***Warning:*** *Beware of the abuse of too many wildcards in your routes, as they may be time consuming to compute when matching a request URI to a route pattern. Better prefer several explicit routes than a too complicated single route.*

Using path variables

Caelyf provides a more convenient way to retrieve the various parts of a request URI, thanks

to path variables.

In a blog application, you want your article to have friendly URLs. For example, a blog post announcing the release of Groovy 1.7-RC-1 could be located at: **`/article/2009/11/27/groovy-17-RC-1-released`**. And you want to be able to reuse the various elements of that URL to pass them in the query string of the Groovlet which is responsible for displaying the article. You can then define a route with path variables as shown in the example below:

```
get "/article/@year/@month/@day/@title", forward: "/article.groovy?
   year=@year&month=@month&day=@day&title=@title"
```

The path variables are of the form **`@something`**, where something is a word (in terms of regular expressions). Here, with our original request URI, the variables will contains the string **`'2009'`** for the **`year`** variable, **`'11'`** for **`month`**, **`'27'`** for **`day`**, and **`'groovy-17-RC-1-released`** for the **`title`** variable. And the final Groovlet URI which will get the request will be **`/WEB-INF/groovy/article.groovy?year=2009&month=11&day=27&title=groovy-17-RC-1-released`**, once the path variable matching is done.

> ***Note:*** *If you want to have optional path variables, you should define as many routes as options. So you would define the following routes to display all the articles published on some year, month, or day:*
>
> ```
> get "/article/@year/@month/@day/@title", forward:
>    "/article.groovy?
>    year=@year&month=@month&day=@day&title=@title"
> get "/article/@year/@month/@day",        forward:
>    "/article.groovy?year=@year&month=@month&day=@day"
> get "/article/@year/@month",             forward:
>    "/article.groovy?year=@year&month=@month"
> get "/article/@year",                    forward:
>    "/article.groovy?year=@year"
> get "/article",                          forward:
>    "/article.groovy"
> ```
>
> *Also, note that routes are matched in order of appearance. So if you have several routes which map an incoming request URI, the first one encountered in the route definition file will win.*

Validating path variables

The routing system also allows you to validate path variables thanks to the usage of a closure. So if you use path variable validation, a request URI will match a route if the route path matches, but also if the closure returns a boolean, or a value which is coercible to a boolean through to the usual *Groovy Truth* rules. Still using our article route, we would like the year to be 4 digits, the month and day 2 digits, and impose no particular constraints on the title path variable, we could define our route as follows:

```
get "/article/@year/@month/@day/@title",
    forward: "/article.groovy?
       year=@year&month=@month&day=@day&title=@title",
    validate: { year ==~ /\d{4}/ && month ==~ /\d{2}/ && day ==~
       /\d{2}/ }
```

> ***Note:*** *Just as the path variables found in the request URI are replaced in the rewritten URL, the path variables are also available inside the body of the closure, so you can apply your validation logic. Here in our closure, we used Groovy's regular expression matching support, but you can use boolean logic that you want, like* ***`year.isNumber()`****, etc.*

In addition to the path variables, you also have access to the **`request`** from within the validation closure. For example, if you wanted to check that a particular attribute is present in the request, like checking a user is registered to access a message board, you could do:

```
get "/message-board",
    forward: "/msgBoard.groovy",
    validate: { request.registered == true }
```

Ignoring certain routes

As a fast path to bypass certain URL patterns, you can use the **`ignore: true`** parameter in your route definition:

```
all "/_ah/**", ignore: true
```

Caching groovlet and template output

Caelyf provides support for caching groovlet and template output, and this be defined through the URL routing system. This caching capability obviously leverages the Redis data structure service of Cloud Foundry. In the definition of your routes, you simply have to add a new named parameter: **`cache`**, indicating the number of seconds, minutes or hours you want the page to be cached. Here are a few examples:

```
get "/news",     forward: "/new.groovy",     cache: 10.minutes
get "/tickers",  forward: "/tickers.groovy", cache: 1.second
get "/download", forward: "/download.gtpl",  cache: 2.hours
```

The duration can be any number (an int) of second(s), minute(s) or hour(s): both plural and singular forms are supported.

It is possible to clear the cache for a given URI if you want to provide a fresher page to your users:

```
memcache.clearCacheForUri('/breaking-news')
```

> ***Note:*** *There are as many cache entries as URIs with query strings. So if you have* ***`/breaking-news`*** *and* ***`/breaking-news?category=politics`****, you will have to clear the cache for both, as* ***Caelyf*** *doesn't track all the query parameters.*

Simple plugin system

Caelyf sports a plugin system which helps you modularize your applications and enable you to share commonalities between **Caelyf** applications.

What a plugin can do for you

A plugin lets you:

- provide additional **groovlets** and **templates**
- contribute new URL **routes**
- add new **categories** to enhance existing classes (like third-party libraries)
- define and bind new **variables in the binding** (the "global" variables available in groovlets and templates)
- provide any kind of **static content**, such as JavaScript, HTML, images, etc.
- add new **libraries** (ie. additional JARs)
- and more generally, let you do any **initialization** at the startup of your application

Possible examples of plugins can:

- provide a "groovy-fied" integration of a third-party library, like nicer JSON support
- create a reusable CRUD administration interface on top of the datastore to easily edit content of all your **Caelyf** applications
- install a shopping cart solution or payment system
- setup a lightweight CMS for editing rich content in a rich-media application
- define a bridge with Web 2.0 applications (Facebook Connect, Twitter authentication)
- and more...

Anatomy of a Caelyf plugin

A plugin is actually just some content you'll drop in your **`war/`** folder, at the root of your **Caelyf** application! This is why you can add all kind of static content, as well as groovlets and templates, or additional JARs in **`WEB-INF/lib`**. Furthermore, plugins don't even need to be external plugins that you install in your applications, but you can just customize your application by using the conventions and capabilities offered by the plugin system. Then, you really just need to have **`/WEB-INF/plugins.groovy`** referencing **`/WEB-INF/plugins/myPluginDescriptor.groovy`**, your plugin descriptor.

In addition to that, you'll have to create a plugin descriptor will allow you to define new binding variables, new routes, new categories, and any initialization code your plugin may need on application startup. This plugin descriptor should be placed in **`WEB-INF/plugins`** and will be a normal groovy script. From this script, you can even access the Cloud Foundry services, which are available in the binding of the script -- hence available somehow as pseudo global variables inside your scripts.

Also, this plugin descriptor script should be referenced in the **`plugins.groovy`** script in **`WEB-INF/`**

Hierarchy

As hinted above, the content of a plugin would look something like the following hierarchy:

```
/
+-- war
    |
    +-- someTemplate.gtpl                          // your templates
    |
    +-- css
    +-- images                                     // your static content
    +-- js
    |
```

```
+-- WEB-INF
    |
    +-- plugins.groovy                      // the list of plugins
    |                                       // descriptors to be installed
    +-- plugins
    |   |
    |   +-- myPluginDescriptor.groovy       // your plugin descriptor
    |
    +-- groovy
    |    |
    |    +-- myGroovlet.groovy              // your groovlets
    |
    +-- includes
    |    |
    |    +-- someInclude.gtpl               // your includes
    |
    +-- classes                             // compiled classes
    |                                       // like categories
    +-- lib
        |
        +-- my-additional-dependency.jar    // your JARs
```

We'll look at the plugin descriptor in a moment, but otherwise, all the content you have in your plugin is actually following the same usual web application conventions in terms of structure, and the ones usually used by **Caelyf** applications (ie. includes, groovlets, etc). The bare minimum to have a plugin in your application is to have a plugin descriptor, like **`/WEB-INF/plugins/myPluginDescriptor.groovy`** in this example, that is referenced in **`/WEB-INF/plugins.groovy`**.

Developing a plugin is just like developing a normal **Caelyf** web application. Follow the usual conventions and describe your plugin in the plugin descriptor. Then afterwards, package it, share it, and install it in your applications.

The plugin descriptor

The plugin descriptor is where you'll be able to tell the **Caelyf** runtime to:

- add new variables in the binding of groovlets and templates
- add new routes to the URL routing system
- define new categories to be applied to enrich APIs (Cloud Foundry services JARs, third-party or your own)
- define before / after request actions
- and do any initialization you may need

Here's what a plugin descriptor can look like:

```groovy
// add imports you need in your descriptor
import net.sf.json.*
import net.sf.json.groovy.*

// add new variables in the binding
binding {
    // a simple string variable
    jsonLibVersion = "2.3"
    // an instance of a class of a third-party JAR
    json = new JsonGroovyBuilder()
}

// add new routes with the usual routing system format
routes {
    get "/json", forward: "/json.groovy"
}

before {
    log.info "Visiting ${request.requestURI}"
```

```
    binding.uri = request.requestURI
    request.message = "Hello"
}

after {
    log.info "Exiting ${request.requestURI}"
}

// install a category you've developped
categories jsonlib.JsonlibCategory

// any other initialization code you'd need
// ...
```

Inside the **`binding`** closure block, you just assign a value to a variable. And this variable will actually be available within your groovlets and templates as implicit variables. So you can reference them with **`${myVar}`** in a template, or use **`myVar`** directly inside a groovlet, without having to declare or retrieve it in any way.

> ***Note:*** *a plugin may overwrite the default* ***Caelyf*** *variable binding, or variable bindings defined by the previous plugin in the initialization chain. In the plugin usage section, you'll learn how to influence the order of loading of plugins.*

Inside the **`routes`** closure block, you'll put the URL routes following the same syntax as the one we explained in the URL routing section.

> ***Note:*** *Contrary to binding variables or categories, the first route that matches is the one which is chosen. This means a plugin cannot overwrite the existing application routes, or routes defined by previous plugins in the chain.*

> ***Important:*** *If your plugins contribute routes, make sure your application has also configured the routes filter, as well as defined a* ***`WEB-INF/routes.groovy`*** *script, otherwise no plugin routes will be present.*

In the **`before`** and **`after`** blocks, you can access the **`request`**, **`response`**, **`log`**, and **`binding`** variables. The logger name is of the form **`caelyf.plugins.myPluginName`**. The **`binding`** variables allows you to update the variables that are put in the binding of Groovlets and templates.

The **`categories`** method call takes a list of classes which are Groovy categories. It's actually just a *varargs* method taking as many classes as you want.

Wherever in your plugin descriptor, you can put any initialization code you may need in your plugin.

> ***Important:*** *The plugins are loaded once, as soon as the first request is served. So your initialization code, adding binding variables, categories and routes, will only be done once per application load.*

Using a plugin

If you recall, we mentioned the **`plugins.groovy`** script. This is a script that lives alongside the **`routes.groovy`** script (if you have one) in **`/WEB-INF`**. If you don't have a **`plugins.groovy`** script, obviously, no plugin will be installed — or at least none of the initialization and configuration done in the various plugin descriptors will ever get run.

This **`plugins.groovy`** configuration file just lists the plugins you have installed and want to use. An example will illustrate how you reference a plugin:

```
install jsonPlugin
```

> ***Note:*** *For each plugin, you'll have an* ***`install`*** *method call, taking as parameter the name of the plugin. This name is actually just the plugin descriptor script name. In this example, this means* ***Caelyf*** *will load* ***`WEB-INF/plugins/jsonPlugin.groovy`****.*

As mentioned previously while talking about the precedence rules, the order with which the plugins are loaded may have an impact on your application or other plugins previously installed and initialized. But hopefully, such conflicts shouldn't happen too often, and this should be resolved easily, as you have full control over the code you're installing through these plugins to make the necessary amendments should there be any.

When you are using two plugins with before / after request actions, the order of execution of these actions also depends on the order in which you installed your plugins. For example, if you have installed **`pluginOne`** first and **`pluginTwo`** second, here's the order of execution of the actions and of the Groovlet or template:

- pluginOne's before action
    - pluginTwo's before action
        - execution of the request
    - pluginTwo's after action
- pluginOne's after action

How to distribute and deploy a plugin

If you want to share a plugin you've worked on, you just need to zip everything that constitutes the plugin. Then you can share this zip, and someone who wishes to install it on his application will just need to unzip it and pickup the various files of that archive and stick them up in the appropriate directories in his/her **Caelyf** **`war/`** folder, and reference that plugin, as explained in the previous section.