Tutorial

The goal of this tutorial is to quickly get you started with using **Gaelyk** to write and deploy Groovy applications on Google App Engine. We'll assume you have already downloaded and installed the Google App Engine SDK on your machine. If you haven't, please do so by reading the instructions from Google.

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 **Gaelyk** servlets
- **`appengine-web.xml`** with the right settings predefined (static file directive)
- a sample Groovlet and template
- the needed JARs (Groovy, Gaelyk and Google App Engine SDK)

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

Table of Content

Setting up your project

Directory layout

We'll follow the directory layout proposed by the **Gaelyk** 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
               |
               +-- appengine-api-1.0-sdk-x.y.z.jar
               +-- appengine-api-labs-x.y.z.jar
               +-- gaelyk-x.y.z.jar
               +-- groovy-all-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:

- **`appengine-web.xml`**: The App Engine specific configuration file we'll detail below.
- **`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, **Gaelyk** and GAE SDK JARs, 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 App Engine or the servlet container expects to find at that specific location.*

The template project comes with initial support for Eclipse project files, allowing you to open the project easily within Eclipse.

> ***Warning:*** *If you're using the Eclipse GAE plugin, be careful with the libraries that the plugin automatically adds to your **`WEB-INF/lib`**. They are not needed for a **Gaelyk** project, and may cause sometimes some strange compilation errors.*

Configuration files

With the directory layout ready, let's have a closer look at the configuration files: the standard **`web.xml`** and App Engine's specific **`appengine-web.xml`**:

appengine-web.xml

```
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
    <!-- Your application ID -->
    <application>myappid</application>

    <version>1</version>

    <!-- If all your templates and groovlets are encoding in UTF-8 --
       >
    <!-- Please specify the settings below, otherwise weird
       characters may appear in your templates -->
    <system-properties>
        <property name="file.encoding" value="UTF-8" />
        <property name="groovy.source.encoding" value="UTF-8" />

        <!-- Define where the logging configuration file should be
           found -->
        <property name="java.util.logging.config.file" value="WEB-
           INF/logging.properties" />
    </system-properties>

    <!-- Uncomment this section if you want your application to be
       able to receive XMPP messages -->
    <!-- Similarily, if you want to receive incoming emails -->
    <!--
    <inbound-services>
        <service>xmpp_message</service>
        <service>mail</service>
    </inbound-services>
```

```
    -->

    <static-files>
        <exclude path="/WEB-INF/**.groovy" />
        <exclude path="**.gtpl" />
    </static-files>
</appengine-web-app>
```

The sole thing which is peculiar here is the fact we're excluding the files with a **`.groovy`** and **`.gtpl`** extensions, as these files are non-static and correspond respectively to the **Gaelyk** Groovlets and templates. We instruct App Engine to not serve these files as mere resource files, like images or stylesheets.

> ***Note:*** *You may decide to use different extensions than* ***`.groovy`*** *and* ***`.gtpl`****, if you prefer to have URLs with extensions which don't leak the underlying technologies being used. Or make sure to use the flexible URL routing system.*

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.gaelyk.GaelykServletContextListener</listener-
           class>
    </listener>

    <!-- The Gaelyk Groovlet servlet -->
    <servlet>
        <servlet-name>GroovletServlet</servlet-name>
        <servlet-class>groovyx.gaelyk.GaelykServlet</servlet-class>
    </servlet>

    <!-- The Gaelyk template servlet -->
    <servlet>
        <servlet-name>TemplateServlet</servlet-name>
        <servlet-class>groovyx.gaelyk.GaelykTemplateServlet</servlet-
           class>
    </servlet>

    <!-- The URL routing filter -->
    <filter>
        <filter-name>RoutesFilter</filter-name>
        <filter-class>groovyx.gaelyk.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>
```

```
        <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 Gaelyk 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 **Gaelyk**. 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* ***Gaelyk*** *provides both view templates and Groovlet controllers, it's advised to use the former for the view and the later for the logic.*

Gaelyk builds on Groovy's own Groovlets and template servlet to add a shorcuts to the App Engine SDK APIs.

> ***Note:*** *You can learn more about Groovy's Groovlets and templates from this article on IBM developerWorks.* ***Gaelyk****'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 App Engine 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, **Gaelyk** also adds ones of his own by injecting specific elements of the Google App Engine SDK:

- **`datastore`** : the Datastore service
- **`memcache`** : the Memcache service
- **`urlFetch`** : the URL Fetch service
- **`mail`** : the Mail service
- **`images`** : the Images service (actually a convenient wrapper class combining both the methods of **`ImagesService`** and **`ImagesServiceFactory`** and implementing the

methods of **`ImagesService`** and **`ImagesServiceFactory`** and implementing the **`ImagesService`** interface)
- **`users`** : the User service
- **`user`** : the currently logged in user (**`null`** if no user logged in)
- **`defaultQueue`** : the default queue
- **`queues`** : a map-like object with which you can access the configured queues
- **`xmpp`** : the Jabber/XMPP service.
- **`blobstore`** : the Blobstore service.
- **`oauth`** : the OAuth service.
- **`namespace`** : the Namespace manager
- **`capabilities`** : the Capabilities service
- **`channel`** : the Channel service
- **`files`** : the File service
- **`backends`** : the Backend service
- **`lifecycle`** : the Lifecycle manager
- **`prospectiveSearch`** : the Prospective search service
- **`localMode`** : a boolean variable which is **`true`** when the application is running in local development mode, and **`false`** when deployed on Google's cloud.
- **`app`** : a map variable with the following keys and values:
  - **`id`** : the application ID (here: gaelyk)
  - **`version`** : the application version (here: 11.354714479246874089)
  - **`env`** : a map with the following keys and values:
    - **`name`** : the environment name (here: Production)
    - **`version`** : the Google App Engine SDK version (here: Google App Engine/1.6.0)
  - **`gaelyk`** : a map with the following keys and values:
    - **`version`** : the version of the **Gaelyk** toolkit used (here: 1.1)

> ***Note:*** *Regarding the* ***`app`*** *variable, this means you can access those values with the following syntax in your groovlets and templates:*
>
> ```
> app.id
> app.version
> app.env.name
> app.env.version
> app.gaelyk.version
> ```

> ***Note:*** *You can learn more about the environment and system properties Google App Engine exposes.*

Thanks to all these variables and services available, you'll be able to access the Google 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 **`@GaelykBindings`** annotation.

```
import groovyx.gaelyk.GaelykBindings

// annotate your class with the transformation
@GaelykBindings
class WeblogService {
    def numberOfComments(post) {
```

```
    def numberOfComments(post) {
        // the datastore service is available
        datastore.execute {
            select count from comments where postId == post.id
        }
    }
}
```

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

Gaelyk 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 instanciate 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 Google App Engine's own services. For instance, the following template will display a different message depending on whether a user is currently logged in or not:

```
<html>
    <body>
        <% if (user) { %>
            <p>You are currently logged in.</p>
        <% } else { %>
            <p>You're not logged in.</p>
        <% } %>
    </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 adivsed, 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

the output stream of 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.gaelyk.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 **`gaelyk.groovlet`** prefix, whereas the template loggers' name starts with **`gaelyk.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`	**`gaelyk.template.myTemplate`**
`/crud/scaffolding.gtpl`	**`gaelyk.template.crud.scaffolding`**
`/WEB-INF/templates/aTemplate.gtpl`	**`gaelyk.template.WEB-INF.templates.aTemplate`**
`/upload.groovy` (ie. **`/WEB-INF/groovy/upload.groovy`**)	**`gaelyk.groovlet.upload`**
`/account/credit.groovy` (ie. **`/WEB-INF/groovy/account/credit.groovy`**)	**`gaelyk.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 **`gaelyk.template.crud.scaffolding`** inherits from **`gaelyk.template.crud`** which inherits in turn from **`gaelyk.template`**, then from **`gaelyk`**. 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 Gaelyk's log level to WARNING, including groovlet's and template's
gaelyk.level = WARNING

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

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

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

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

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

```
import groovyx.gaelyk.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 **`gaelyk.routesfilter`** and **`gaelyk.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

Flexible URL routing

Gaelyk 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.gaelyk.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.gaelyk.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
- use the available GAE services in the script (for instance, creating routes from records

from the datastore)
- cache the output of groovlets and templates pointed by that route for a specified period of time
- specify a handler for incoming email messages
- specify a handler for incoming jabber messages

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,* ***Gaelyk*** *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,* ***Gaelyk*** *checks whether a more recent route definition file exists. However, once deployed on the Google App Engine cloud, 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.*

Incoming email and jabber messages

Two special routing rules exist for defining handlers dedicated to receiving incoming email messages and jabber messages.

```
email  to: "/receiveEmail.groovy"
jabber to: "/receiveJabber.groovy"
```

jabber chat, to: "/receiveJabber.groovy" // synonym of jabber to: "..." // for Jabber subscriptions jabber subscription, to: "/subs.groovy" // for Jabber user presence notifications jabber presence, to: "/presence.groovy"

> ***Note:*** *Those two notations are actually equivalent to:*
>
> ```
> post "/_ah/mail/*", forward: "/receiveEmail.groovy"
> post "/_ah/xmpp/message/chat/", forward:
>    "/receiveJabber.groovy"
> ```
>
> *Should upcoming App Engine SDK versions change the URLs, you would still be able*

> *to define routes for those handlers, till a new version of **Gaelyk** is released with the newer paths.*

> ***Note:** Make sure to read the sections on incoming email messages and incoming jabber messages.*

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.*

Warmup requests

When an application running on a production instance receives too many incoming requests, App Engine will spawn a new server instance to serve your users. However, the new incoming requests were routed directly to the new instance, even if the application wasn't yet fully initialized for serving requests, and users would face the infamous "loading request" issue, with long response times, as the application needed to be fully initialized to be ready to serve those requests. Thanks to "warmup requests", Google App Engine does a best effort at honoring the time an application needs to be fully started, before throwing new incoming requests to that new instance.

Warmup requests are enabled by default, and new traffic should be directed to new application instances only when the following artefacts are initialized:

- Servlets configured with **`load-on-startup`** and their **`void init(ServletConfig)`** method was called.
- Servlet filters have had their **`void init(FilterConfig)`** method was called.
- Servlet context listeners have had their **`void contextInitialized(ServletContextEvent)`** method was called.

> ***Note:** Please have a look at the documentation regarding "warmup requests". Please also note that you can also enable billing and activate an option to reserve 3 warm JVMs ready to serve your requests.*

So to benefit from "warmup requests", the best approach is to follow those standard initialization procedures. However, you can also define a special Groovlet handler for those warmup

requests through the URL routing mechanism. Your Groovlet will be responsible for the initialization phase your application may be needing. To define a route for the "warmup requests", you can procede as follows:

```
all "/_ah/warmup", forward: "/myWarmupRequestHandler.groovy"
```

Using path variables

Gaelyk 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 }
```

Capability-aware routing

With Google App Engine's capability service, it is possible to programmatically decide what your application is supposed to be doing when certain services aren't functionning as they should be or are scheduled for maintenance. For instance, you can react upon the unavailability of the datastore, etc. With this mechanism available, it is also possible to customize your routes to cope with the various statuses of the available App Engine services.

> ***Note:*** *Please make sure to have a look at the capabilities support provided by* ***Gaelyk****.*

To leverage this mechanism, instead of using a simple string representing the redirect or forward destination of a route, you can also use a closure with sub-rules defining the routing, depending on the status of the services:

```
import static com.google.appengine.api.capabilities.Capability.*
import static
   com.google.appengine.api.capabilities.CapabilityStatus.*

get "/update", forward: {
    to "/update.groovy"
    to("/maintenance.gtpl").on(DATASTORE)       .not(ENABLED)
    to("/readonly.gtpl")    .on(DATASTORE_WRITE).not(ENABLED)
}
```

In the example above, we're passing a closure to the forward parameter. There is a mandatory default destination defined: **`/update.groovy`**, that is chosen if no capability-aware sub-rule matches.

> ***Important:*** *The sub-rules are checked in the order they are defined: so the first one matching will be applied. If none matches, the default destination will be used.*

The sub-rules are represented in the form of chained method calls:

- A destination is defined with the **`to("/maintenance.gtpl")`** method.

- Then, an **`on(DATASTORE)`** method tells which capability should the rule be checked against.
- Eventually, the **`not(ENABLED)`** method is used to check if the **`DATASTORE`** is **`not`** in the status **`ENABLED`**.

> ***Tip:*** *If you're using Groovy 1.8-beta-2 and beyond, you'll be able to use an even nicer syntax, with fewer punctuation marks:*
>
> ```
> import static
>    com.google.appengine.api.capabilities.Capability.*
> import static
>    com.google.appengine.api.capabilities.CapabilityStatus.*
>
> get "/update", forward: {
>     to "/update.groovy"
>     to "/maintenance.gtpl" on DATASTORE       not ENABLED
>     to "/readonly.gtpl"    on DATASTORE_WRITE not ENABLED
> }
> ```

The last method of the chain can be either **`not()`**, as in our previous examples, or **`is()`**. For example, you can define a sub-rule for the case where a scheduled maintenance window is planned:

```
// using Groovy-1.8-beta-2+ syntax:
to "/urlFetchMaintenance.gtpl" on URL_FETCH is SCHEDULED_MAINTENANCE
```

The following capabilities are available, as defined as constants in the Capability class:

- BLOBSTORE
- DATASTORE
- DATASTORE_WRITE
- IMAGES
- MAIL
- MEMCACHE
- TASKQUEUE
- URL_FETCH
- XMPP

The available status capabilities, as defined on the CapabilityStatus enum, are as follows:

- ENABLED
- DISABLED
- SCHEDULED_MAINTENANCE
- UNKNOWN

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

Gaelyk provides support for caching groovlet and template output, and this be defined through the URL routing system. This caching capability obviously leverages the Memcache service of

Google App Engine. 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.

> ***Note:*** *byte arrays (the content to be cached) and strings (the URI, the content-type and last modified information) are stored in Memcache, and as they are simple types, they should even survive Google App Engine loading requests.*

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* ***Gaelyk*** *doesn't track all the query parameters.*

Namespace scoped routes

Another feature of the URL routing system, with the combination of Google App Engine's namespace handling support, is the ability to define a namespace, for a given route. This mechanism is particularly useful when you want to segregate data for a user, a customer, a company, etc., i.e. as soon as you're looking for making your application multitenant. Let's see this in action with an example:

```
post "/customer/@cust/update", forward: "/customerUpdate.groovy?
   cust=@cust", namespace: { "namespace-$cust" }
```

For the route above, we want to use a namespace per customer. The **`namespace`** closure will be called for each request to that route, returning the name of the namespace to use, in the scope of that request. If the incoming URI is **`/customer/acme/update`**, the resulting namespace used for that request will be **`namespace-acme`**.

> ***Note:*** *Make sure to have a look at the namespace support also built-in* ***Gaelyk****.*

Google App Engine specific shortcuts

Google App Engine specific shortcuts

In addition to providing direct access to the App Engine services, **Gaelyk** also adds some syntax sugar on top of these APIs. Let's review some of these improvements.

Improvements to the low-level datastore API

Although it's possible to use JDO and JPA in Google App Engine, **Gaelyk** also lets you use the low-level raw API for accessing the datastore, and makes the **`Entity`** class from that API a bit more Groovy-friendly.

Using `Entity`s as maps or POJOs/POGOs

> ***Note:*** *POGO stands for Plain Old Groovy Object.*

When you use the **`Entity`** class from Java, you have to use methods like **`setProperty()`** or **`getProperty()`** to access the properties of your **`Entity`**, making the code more verbose than it needs to be (at least in Java). Ultimately, you would like to be able to use this class (and its instances) as if they were just like a simple map, or as a normal Java Bean. That's what **Gaelyk** proposes by letting you use the subscript operator just like on maps, or a normal property notation. The following example shows how you can access **`Entity`**s:

```
import com.google.appengine.api.datastore.Entity

Entity entity = new Entity("person")

// subscript notation, like when accessing a map
entity['name'] = "Guillaume Laforge"
println entity['name']

// normal property access notation
entity.age = 32
println entity.age
```

> ***Note:*** *For string properties, Google App Engine usually distinguishes between strings longer or shorter than 500 characters. Short strings are just mere Java strings, while longer strings (>500 chars) should be wrapped in a **`Text`** instance. **Gaelyk** shields you from taking care of the difference, and instead, when using the two notations above, you just have to deal with mere Java strings, and don't need to use the **`Text`** class at all.*

Some properties of your entities can be unindexed, meaning that they can't be used for your search criteria. This may be the case for long text properties, for example a bio of a person, etc. The property notation or subscript notation use normal indexed properties. If you want to set unindexed properties, you can use the **`unindexed`** shorcut:

```
import com.google.appengine.api.datastore.Entity

Entity entity = new Entity("person")

entity.name = "Guillaume Laforge"
entity.unindexed.bio = "Groovy Project Manager..."
entity.unindexed['address'] = "Very long address..."
```

A handy mechanism exists to assign several properties at once, on your entities, using the **<<** (left shift) operator. This is particularly useful when you have properties coming from the request, in the **`params`** map variable. You can the do the following to assign all the key/values

in the map as properties on your entity:

```
// the request parameters contain a firstname, lastname and age
   key/values:
// params = [firstname: 'Guillaume', lastname: 'Laforge', title:
   'Groovy Project Manager']

Entity entity = new Entity("person")

entity << params

assert entity.lastname == 'Laforge'
assert entity.firstname == 'Guillaume'
assert entity.title == 'Groovy Project Manager'

// you can also select only the key/value pairs you'd like to set on
   the entity
// thanks to Groovy's subMap() method, which will create a new map
   with just the keys you want to keep
entity << params.subMap(['firstname', 'lastname'])
```

***Note: Gaelyk** adds a few converter methods to ease the creation of instances of some GAE SDK types that can be used as properties of entities, using the **as** operator:*

```
"foobar@gmail.com" as Email
"foobar@gmail.com" as JID

"http://www.google.com" as Link
new URL("http://gaelyk.appspot.com") as Link

"+33612345678" as PhoneNumber
"50 avenue de la Madeleine, Paris" as PostalAddress

"groovy" as Category

32 as Rating
"32" as Rating

"long text" as Text

"some byte".getBytes() as Blob
"some byte".getBytes() as ShortBlob

"foobar" as BlobKey

[45.32, 54.54] as GeoPt
```

Converting beans to entities and back

The mechanism explained above with type conversions (actually called "coercion") is also available and can be handy for converting between a concrete bean and an entity. Any POJO or POGO can thus be converted into an **`Entity`**, and you can also convert an **`Entity`** to a POJO or POGO.

```
// given a POJO
class Person {
    String name
    int age
}
```

```
}

def e1 = new Entity("Person")
e1.name = "Guillaume"
e1.age = 33

// coerce an entity into a POJO
def p1 = e1 as Person

assert e1.name == p1.name
assert e1.age == p1.age

def p2 = new Person(name: "Guillaume", age: 33)
// coerce a POJO into an entity
def e2 = p2 as Entity

assert p2.name == e2.name
assert p2.age == e2.age
```

> ***Note:*** *The POJO/POGO class* ***`simpleName`*** *property is used as the entity kind. So for example, if the* ***`Person`*** *class was in a package* ***`com.foo`****, the entity kind used would be* ***`Person`****, not the fully-qualified name. This is the same default strategy that Objectify is using.*

Further customization of the coercion can be achieved by using 3 annotations on your classes:

- **`@Key`** to specify that a particular property or getter method should be used as the key for the entity (should be a String or a long)
- **`@Unindexed`** for properties or getter methods that should be set as unindexed (ie. on which no queries can be done)
- **`@Ignore`** for properties or getter methods that should be ignored and not persisted

Here's an example of a **`Person`** bean, whose key is a string login, whose biography should be unindexed, and whose full name can be ignored since it's a computed property:

```
import groovyx.gaelyk.datastore.Key
import groovyx.gaelyk.datastore.Unindexed
import groovyx.gaelyk.datastore.Ignore

class Person {
    @Key String login
    String firstName
    String lastName
    @Unindexed String bio
    @Ignore String getFullName() { "$firstName $lastName" }
}
```

> ***Note:*** *In turn, with this feature, you have a lightweight object/entity mapper. However, remember it's a simplistic solution for doing object/entity mapping, and this solution doesn't take into accounts relationships and such. If you're really interested in a fully featured mapper, you should have a look at Objectify or Twig.*

List to `Key` conversion

Another coercion mechanism that you can take advantage of, is to use a list to **`Key`** conversion, instead of using the more verbose **`KeyFactory.createKey()`** methods:

```
[parentKey, 'address', 333] as Key
```

```
[parentKey, 'address', 'name'] as Key
['address', 444] as Key
['address', 'name'] as Key
```

Added `save()` and `delete()` methods on `Entity`

In the previous sub-section, we've created an **`Entity`**, but we need to store it in Google App Engine's datastore. We may also wish to delete an **`Entity`** we would have retrieved from that datastore. For doing so, in a *classical* way, you'd need to call the **`save()`** and **`put()`** methods from the **`DataService`** instance. However, **Gaelyk** dynamically adds a **`save()`** and **`delete()`** method on **`Entity`**:

```
def entity = new Entity("person")
entity.name = "Guillaume Laforge"
entity.age = 32

entity.save()
```

Afterwards, if you need to delete the **`Entity`** you're working on, you can simply call:

```
entity.delete()
```

Added `delete()` and `get()` method on `Key`

Sometimes, you are dealing with keys, rather than dealing with entities directly — the main reaons being often for performance sake, as you don't have to load the full entity. If you want to delete an element in the datastore, when you just have the key, you can do so as follows:

```
someEntityKey.delete()
```

Given a **`Key`**, you can get the associated entity with the **`get()`** method:

```
Entity e = someEntityKey.get()
```

And if you have a list of entities, you can get them all at once:

```
def map = [key1, key2].get()

// and then access the returned entity from the map:
map[key1]
```

Converting `Key` to an encoded `String` and vice-versa

When you want to store a **`Key`** as a string or pass it as a URL parameter, you can use the **`KeyFactory`** methods to encode / decode keys and their string representations. **Gaelyk** provides two convenient coercion mechanisms to get the encoded string representation of a key:

```
def key = ['addresses', 1234] as Key
def encodedKey = key as String
```

And to retrieve the key from its encoded string representation:

```
def encodedKey = params.personKey  // the encoded string
   representation of the key
def key = encodedKey as Key
```

```
def key = encodeKey as key
```

Added `withTransaction()` method on the datastore service

Last but not least, if you want to work with transactions, instead of using the **`beginTransaction()`** method of **`DataService`**, then the **`commit()`** and **`rollback()`** methods on that **`Transaction`**, and doing the proper transaction handling yourself, you can use the **`withTransaction()`** method that **Gaelyk** adds on **`DataService`** and which takes care of that boring task for you:

```
datastore.withTransaction {
    // do stuff with your entities within the transaction
}
```

The **`withTransaction()`** method takes a closure as the sole parameter, and within that closure, upon its execution by **Gaelyk**, your code will be in the context of a transaction.

Added `get()` methods on the datastore service

To retrieve entities from the datastore, you can use the **`datastore.get(someKey)`** method, and pass it a **`Key`** you'd have created with **`KeyFactory.createKey(...)`**: this is a bit verbose, and **Gaelyk** proposes additional **`get()`** methods on the datastore service, which do the key creation for you:

```
Key pk = ... // some parent key
datastore.get(pk, 'address', 'home') // by parent key, kind and name
datastore.get(pk, 'address', 1234)   // by parent key, kind and id

datastore.get('animal', 'Felix')     // by kind and name
datastore.get('animal', 2345)        // by kind and id
```

This mechanism also works with the asynchronous datastore, as **Gaelyk** wraps the **`Future<Entity>`** transparently, so you don't have to call **`get()`** on the future:

```
Key pk = ... // some parent key
datastore.async.get(pk, 'address', 'home') // by parent key, kind
   and name
datastore.async.get(pk, 'address', 1234)   // by parent key, kind
   and id

datastore.async.get('animal', 'Felix')     // by kind and name
datastore.async.get('animal', 2345)        // by kind and id
```

> ***Note:*** *When you have a* ***`Future<Entity> f`****, when you call* ***`f.someProperty`****,* ***Gaelyk*** *will actually lazily call* ***`f.get().someProperty`****, making the usage of the future transparent. However, note it only works for properties, it doesn't work for method call on futures, where you will have to call* ***`get()`*** *first. This transparent handling of future properties is working for all* ***`Future`****s, not just* ***`Future<Entity>`****.*

Querying

With the datastore API, to query the datastore, the usual approach is to create a **`Query`**, prepare a **`PreparedQuery`**, and retrieve the results as a list or iterator. Below you will see an example of queries used in the Groovy Web Console to retrieve scripts written by a given author, sorted by descending date of creation:

```
import com.google.appengine.api.datastore.*
```

```
import static
   com.google.appengine.api.datastore.FetchOptions.Builder.*

// query the scripts stored in the datastore
// "savedscript" corresponds to the entity table containing the
   scripts' text
def query = new Query("savedscript")

// sort results by descending order of the creation date
query.addSort("dateCreated", Query.SortDirection.DESCENDING)

// filters the entities so as to return only scripts by a certain
   author
query.addFilter("author", Query.FilterOperator.EQUAL, params.author)

PreparedQuery preparedQuery = datastore.prepare(query)

// return only the first 10 results
def entities = preparedQuery.asList( withLimit(10) )
```

Fortunately, **Gaelyk** provides a query DSL for simplifying the way you can query the datastore. Here's what it looks like with the query DSL:

```
def entities = datastore.execute {
    select all from savedscript
    sort desc by dateCreated
    where author == params.author
    limit 10
}
```

Let's have a closer look at the syntax supported by the DSL. There are two methods added dynamically to the datastore: **`query{}`** and **`execute{}`**. The former allow you to create a **`Query`** that you can use then to prepare a **`PreparedQuery`**. The latter is going further as it executes the query to return a single entity, a list, a count, etc.

Creating queries

You can create a **`Query`** with the **`datastore.query{}`** method. The closure argument passed to this method supports the verbs **`select`**, **`from`**, **`where/and`** and **`sort`**. Here are the various options of those verbs:

```
// select the full entity with all its properties
select all
// return just the keys of the entities matched by the query
select keys

// specify the entity kind to search into
from entityKind

// specify that entities searched should be child of another entity
// represented by its key
ancestor entityKey

// add a filter operation
// operators allowed are: <, <=, ==, !=, >, >=, in
where propertyName <  value
where propertyName <= value
where propertyName == value
where propertyName != value
where propertyName >= value
where propertyName >  value
where propertyName in listOfValues
```

```
where propertyName in listOfValues

// you can use "and" instead of "where" to add more where clauses

// ascending sorting
sort asc  by propertyName
// descending sorting
sort desc by propertyName
```

> ***Notes:***
>
> - *The entity kind of the **`from`** verb and the property name of the **`where`** verb and **`sort/by`** verbrs are actually mere strings, but you don't need to quote them.*
> - *Also, for the **`where`** clause, be sure to put the property name on the left-hand-side of the comparison, and the compared value on the right-hand-side of the operator.*
> - *When you need more than one **`where`** clause, you can use **`and`** which is a synonym of **`where`**.*
> - *You can omit the **`select`** part of the query if you wish: by default, it will be equivalent to **`select all`**.*
> - *It is possible to put all the verbs of the DSL on a single line (thanks to Groovy 1.8 command chains notation), or split across several lines as you see fit for readability or compactness.*

Executing queries

You can use the **`datastore.execute{}`** call to execute the queries, or the **`datastore.iterate{}`** call if you want to get the results in the form of an iterator. The **`select`** verb also provides additional values. The **`from`** verb allows to specify a class to coerce the results to a POGO. In addition, you can specify the **`FetchOptions`** with additional verbs like: **`limit`**, **`offset`**, **`range`**, **`chunkSize`**, **`fetchSize`** **`startAt`**, **`endAt`**

```
// select the full entity with all its properties
select all
// return just the keys of the entities matched by the query
select keys
// return one single entity if the query really returns one single
   result
select single
// return the count of entities matched by the query
select count

// from an entity kind
from entityKind
// specify the entity kind as well as a type to coerce the results
   to
from entityKind as SomeClass

// specify that entities searched should be child of another entity
// represented by its key
ancestor entityKey

where propertyName <  value
where propertyName <= value
where propertyName == value
where propertyName != value
where propertyName >= value
where propertyName >  value

where propertyName in listOfValues
```

```
// you can use "and" instead of "where" to add more where clauses

// ascending sorting
sort asc  by propertyName
// descending sorting
sort desc by propertyName

// limit to only 10 results
limit 10
// return the results starting from a certain offset
offset 100
// range combines offset and limit together
range 100..109

// fetch and chunk sizes
fetchSize 100
chunkSize 100

// cursor handling
startAt cursorVariable
startAt cursorWebSafeStringRepresentation
endAt cursorVariable
endAt cursorWebSafeStringRepresentation
```

> ***Notes:*** *If you use the* ***`from addresses as Address`*** *clause, specifying a class to coerce the results into, if your* ***`where`*** *and* ***`and`*** *clauses use properties that are not present in the target class, a* ***`QuerySyntaxException`*** *will be thrown.*

Asynchronous datastore

In addition to the "synchronous" datastore service, the App Engine SDK also provides an AsynchrnousDatastoreService. You can retrieve the asynchronous service with the **`datastore.async`** shortcut.

Gaelyk adds a few methods on entities and keys that leverage the asynchronous service:

- **`entity.asyncSave()`** returns a **`Future<Key>`**
- **`entity.asyncDelete()`** returns a **`Future<Void>`**
- **`key.asyncDelete()`** returns a **`Future<Void>`**

Datastore metadata querying

The datastore contains some special entities representing useful metadata, like the available kinds, namespaces and properties. **Gaelyk** provides shortcuts to interrogate the datastore for such entity metadata.

Namespace querying

```
// retrieve the list of namespaces (as a List<Entity>)
def namespaces = datastore.namespaces

// access the string names of the namespaces
def namespaceNames = namespaces.key.name

// if you want only the first two
datastore.getNamespaces(FetchOptions.Builder.withLimit(2))

// if you want to apply further filtering on the underlying
   datastore query
datastore.getNamespaces(FetchOptions.Builder.withLimit(2)) { Query
```

```
   query ->
    // apply further filtering on the query parameter
}
```

Kind querying

```
// retrieve the list of entity kinds (as a List<Entity>)
def kinds = datastore.kinds

// get only the string names
def kindNames = kinds.key.name

// get the first kind
datastore.getKinds(FetchOptions.Builder.withLimit(10))

// futher query filtering:
datastore.getKinds(FetchOptions.Builder.withLimit(10)) { Query query
   ->
    // apply further filtering on the query parameter
}
```

Properties querying

```
// retrieve the list of entity properties (as a List<Entity>)
def props = datastore.properties

// as for namespaces and kinds, you can add further filtering
datastore.getProperties(FetchOptions.Builder.withLimit(10)) { Query
   query ->
    // apply further filtering on the query parameter
}

// if you want to retrive the list of properties for a given entity
   kind,
// for an entity Person, with two properties name and age:
def entityKindProps = datastore.getProperties('Person')
// lists of entity names
assert entityKindProps.key.parent.name == ['Person', 'Person']
// list of entity properties
assert entityKindProps.key.name == ['name', 'age']
```

The task queue API shortcuts

Google App Engine SDK provides support for "task queues". An application has a default queue, but other queues can be added through the configuration of a **`queue.xml`** file in **`/WEB-INF`**.

> ***Note:*** *You can learn more about queues and task queues, and how to configure them on the online documentation.*

In your Groovlets and templates, you can access the default queue directly, as it is passed into the binding:

```
// access the default queue
defaultQueue
```

You can access the queues either using a subscript notation or the property access notation:

```
// access a configured queue named "dailyEmailQueue" using the
   subscript notation
queues['dailyEmailQueue']

// or using the property access notation
queues.dailyEmailQueue

// you can also access the default queue with:
queues.default
```

To get the name of a queue, you can call the provided **`getQueueName()`** method, but **Gaelyk** provides also a **`getName()`** method on Queue so that you can write **`queue.name`**, instead of the more verbose **`queue.getQueueName()`** or **`queue.queueName`**, thus avoid repetition of queue.

For creating tasks and submitting them on a queue, with the SDK you have to use the **`TaskOptions.Builder`**. In addition to this builder approach, **Gaelyk** provides a shortcut notation for adding tasks to the queue using named arguments:

```
// add a task to the queue
queue.add countdownMillis: 1000, url: "/task/dailyEmail",
    taskName: "dailyNewsletter",
    method: 'PUT', params: [date: '20101214'],
    payload: content, retryOptions:
       RetryOptions.Builder.withDefaults()
```

There is also a variant with an overloaded **<<** operator:

```
// add a task to the queue
queue << [
    countdownMillis: 1000, url: "/task/dailyEmail",
    taskName: "dailyNewsletter",
    method: 'PUT', params: [date: '20101214'],
    payload: content,
    retryOptions: [
        taskRetryLimit: 10,
        taskAgeLimitSeconds: 100,
        minBackoffSeconds: 40,
        maxBackoffSeconds: 50,
        maxDoublings: 15
    ]
]
```

Email support

New `send()` method for the mail service

Gaelyk adds a new **`send()`** method to the mail service, which takes *named arguments*. That way, you don't have to manually build a new message yourself. In your Groovlet, for sending a message, you can do this:

```
mail.send from: "app-admin-email@gmail.com",
        to: "recipient@somecompany.com",
        subject: "Hello",
        textBody: "Hello, how are you doing? -- MrG",
        attachment: [data: "Chapter 1, Chapter 2".bytes, fileName:
           "outline.txt"]
```

Similarily, a **`sendToAdmins()`** method was added to, for sending emails to the administrators

Similarly, a **`sendToAdmins()`** method was added to, for sending emails to the administrators of the application.

> ***Note:*** *There is a* ***`sender`*** *alias for the* ***`from`*** *attribute. And instead of a* ***`textBody`*** *attribute, you can send HTML content with the* ***`htmlBody`*** *attribute.*

> ***Note:*** *There are two attachment attributes:* ***`attachment`*** *and* ***`attachments`****.*
>
> - ***`attachment`*** *is used for when you want to send just one attachment. You can pass a map with a* ***`data`*** *and a* ***`fileName`*** *keys. Or you can use an instance of* ***`MailMessage.Attachment`****.*
> - ***`attachments`*** *lets you define a list of attachments. Again, either the elements of that list are maps of* ***`data`*** */* ***`fileName`*** *pairs, or instances of* ***`MailMessage.Attachment`****.*

Incoming email messages

Your applications can also receive incoming email messages, in a similar vein as the incoming XMPP messaging support. To enable incoming email support, you first need to update your **`appengine-web.xml`** file as follows:

```
<inbound-services>
    <service>mail</service>
</inbound-services>
```

In your **`web.xml`** file, you can eventually add a security constraint on the web handler that will take care of treating the incoming emails:

```
...
<!-- Only allow the SDK and administrators to have access to the
   incoming email endpoint -->
<security-constraint>
    <web-resource-collection>
        <url-pattern>/_ah/mail/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>
...
```

You need to define a Groovlet handler for receiving the incoming emails with a special route definition, in your **`/WEB-INF/routes.groovy`** configuration file:

```
email to: "/receiveEmail.groovy"
```

> ***Remark:*** *You are obviously free to change the name and path of the Groovlet.*

All the incoming emails will be sent as MIME messages through the request of your Groovlet. To parse the MIME message, you'll be able to use the **`parseMessage(request)`** method on the mail service injected in the binding of your Groovlet, which returns a **`javax.mail.MimeMessage`** instance:

```
def msg = mail.parseMessage(request)

log.info "Subject ${msg.subject}, to ${msg.allRecipients.join(',
```

```
    ')}, from ${msg.from[0]}"
```

XMPP/Jabber support

Your application can send and receive instant messaging through XMPP/Jabber.

> ***Note:*** *You can learn more about XMPP support on the online documentation.*

Sending messages

Gaelyk provides a few additional methods to take care of sending instant messages, get the presence of users, or to send invitations to other users. Applications usually have a corresponding Jabber ID named after your application ID, such as **`yourappid@appspot.com`**. To be able to send messages to other users, your application will have to invite other users, or be invited to chat. So make sure you do so for being able to send messages.

Let's see what it would look like in a Groovlet for sending messages to a user:

```
String recipient = "someone@gmail.com"

// check if the user is online
if (xmpp.getPresence(recipient).isAvailable()) {
    // send the message
    def status = xmpp.send(to: recipient, body: "Hello, how are
       you?")

    // checks the message was successfully delivered to all the
       recipients
    assert status.isSuccessful()
}
```

Gaelyk once again decorates the various XMPP-related classes in the App Engine SDK with new methods:

- on **`XMPPService`**'s instance
  - **`SendResponse send(Map msgAttr)`** : more details on this method below
  - **`void sendInvitation(String jabberId)`** : send an invitation to a user
  - **`sendInvitation(String jabberIdTo, String jabberIdFrom)`** : send an invitation to a user from a different Jabber ID
  - **`Presence getPresence(String jabberId)`** : get the presence of this particular user
  - **`Presence getPresence(String jabberIdTo, String jabberIdFrom)`** : same as above but using a different Jabber ID for the request
- on **`Message`** instances
  - **`String getFrom()`** : get the Jabber ID of the sender of this message
  - **`GPathResult getXml()`** : get the XmlSlurper parsed document of the XML payload
  - **`List<String> getRecipients()`** : get a list of Strings representing the Jabber IDs of the recipients
- on **`SendResponse`** instances
  - **`boolean isSuccessful()`** : checks that all recipients received the message

To give you a little more details on the various attributes you can use to create messages to be sent, you can pass the following attributes to the **`send()`** method of **`XMPPService`**:

- **`body`** : the raw text content of your message
- **`xml`** : a closure representing the XML payload you want to send

- **`to`** : contains the recipients of the message (either a String or a List of String)
- **`from`** : a String representing the Jabber ID of the sender
- **`type`** : either an instance of the **`MessageType`** enum or a String (**`'CHAT'`**, **`'ERROR'`**, **`'GROUPCHAT'`**, **`'HEADLINE'`**, **`'NORMAL'`**)

> ***Note:*** ***`body`*** *and* ***`xml`*** *are exclusive, you can't specify both at the same time.*

We mentioned the ability to send XML payloads, instead of normal chat messages: this functionality is particularly interesting if you want to use XMPP/Jabber as a communication transport between services, computers, etc. (ie. not just real human beings in front of their computer). We've shown an example of sending raw text messages, here's how you could use closures in the **`xml`** to send XML fragments to a remote service:

```
String recipient = "service@gmail.com"

// check if the service is online
if (xmpp.getPresence(recipient).isAvailable()) {
    // send the message
    def status = xmpp.send(to: recipient, xml: {
        customers {
            customer(id: 1) {
                name 'Google'
            }
        }
    })

    // checks the message was successfully delivered to the service
    assert status.isSuccessful()
}
```

> ***Implementation detail:*** *the closure associated with the* ***`xml`*** *attribute is actually passed to an instance of* ***`StreamingMarkupBuilder`*** *which creates an XML stanza.*

Receiving messages

It is also possible to receive messages from users. For that purpose, **Gaelyk** lets you define a Groovlet handler that will be receiving the incoming messages. To enable the reception of messages, you'll have to do two things:

- add a new configuration fragment in **`/WEB-INF/appengine-web.xml`**
- add a route for the Groovlet handler in **`/WEB-INF/routes.groovy`**

As a first step, let's configure **`appengine-web.xml`** by adding this new element:

```
<inbound-services>
    <service>xmpp_message</service>
</inbound-services>
```

Similarily to the incoming email support, you can define security constraints:

```
...
<!-- Only allow the SDK and administrators to have access to the
   incoming jabber endpoint -->
<security-constraint>
    <web-resource-collection>
        <url-pattern>/_ah/xmpp/message/chat/</url-pattern>
    </web-resource-collection>
```

```
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>
...
```

Then let's add the route definition in **routes.groovy**:

```
jabber to: "/receiveJabber.groovy"
```

Alternatively, you can use the longer version:

```
jabber chat, to: "/receiveJabber.groovy"
```

> ***Remark:*** *You are obviously free to change the name and path of the Groovlet.*

All the incoming Jabber/XMPP messages will be sent through the request of your Groovlet. Thanks to the **parseMessage(request)** method on the **xmpp** service injected in the binding of your Groovlet, you'll be able to access the details of a **Message** instance, as shown below:

```
def message = xmpp.parseMessage(request)

log.info "Received from ${message.from} with body ${message.body}"

// if the message is an XML document instead of a raw string message
if (message.isXml()) {
    // get the raw XML string
    message.stanza

    // or get a document parsed with XmlSlurper
    message.xml
}
```

XMPP presence handling

To be notified of users' presence, you should first configure **appengine-web.xml** to specify you want to activate the incoming presence service:

```
<inbound-services>
    <service>xmpp_presence</service>
</inbound-services>
```

Then, add a special route definition in **routes.groovy**:

```
jabber presence, to: "/presence.groovy"
```

> ***Remark:*** *You are obviously free to change the name and path of the Groovlet handling the presence requests.*

Now, in your **presence.groovy** Groovlet, you can call the overriden **XMPPService#parsePresence** method:

```
// parse the incoming presence from the request
def presence = xmpp.parsePresence(request)

log.info "${presence.fromJid.id} is ${presence.available ? '' :
```

```
    'not' available"
```

XMPP subscription handling

To be notified of subscriptions, you should first configure **`appengine-web.xml`** to specify you want to activate the incoming subscription service:

```
<inbound-services>
    <service>xmpp_subscribe</service>
</inbound-services>
```

Then, add a special route definition in **`routes.groovy`**:

```
jabber subscription, to: "/subscription.groovy"
```

> ***Remark:*** *You are obviously free to change the name and path of the Groovlet handling the subscription requests.*

Now, in your **`subscription.groovy`** Groovlet, you can call the overriden **`XMPPService#parseSubscription`** method:

```
// parse the incoming subscription from the request
def subscription = xmpp.parseSubscription(request)

log.info "Subscription from ${subscription.fromJid.id}:
   ${subscription.subscriptionType}}"
```

Enhancements to the Memcache service

Gaelyk provides a few additional methods to the Memcache service, to get and put values in the cache using Groovy's natural subscript notation, as well as for using the **`in`** keyword to check when a key is present in the cache or not.

```
class Country implements Serializable { String name }

def countryFr = new Country(name: 'France')

// use the subscript notation to put a country object in the cache,
   identified by a string
// (you can also use non-string keys)
memcache['FR'] = countryFr

// check that a key is present in the cache
if ('FR' in memcache) {
    // use the subscript notation to get an entry from the cache
       using a key
    def countryFromCache = memcache['FR']
}
```

> ***Note:*** *Make sure the objects you put in the cache are serializable. Also, be careful with the last example above as the `'FR'` entry in the cache may have disappeared between the time you do the `if (... in ...)` check and the time you actually retrieve the value associated with the key from memcache.*

Asynchronous Memcache service

The Memcache service is synchronous, but App Engine also proposes an asynchronous Memcache service that you can access by calling the **`async`** property on the Memcache service instance:

```
memcache.async.put(key, value)
```

> ***Note:*** *Additionally, the usual property notation and subscript access notation are also available.*

Closure memoization

As Wikipedia puts it, memoization is an *optimization technique used primarily to speed up computer programs by having function calls avoid repeating the calculation of results for previously-processed inputs*. **Gaelyk** provides such a mechanism for closures, storing invocation information (a closure call with its arguments values) in memcache.

An example, if you want to avoid computing expansive operations (like repeatedly fetching results from the datastore) in a complex algorithm:

```
Closure countEntities = memcache.memoize { String kind ->
    datastore.prepare( new Query(kind) ).countEntities()
}

// the first time, the expensive datastore operation will be
   performed and cached
def totalPics = countEntities('photo')

/* add new pictures to the datastore */

// the second invocation, the result of the call will be the same as
   before, coming from the cache
def totalPics2 = countEntities('photo')
```

> ***Note:*** *Invocations are stored in memcache only for up to the 30 seconds request time limit of App Engine.*

Enhancements related to the Blobstore and File services

Gaelyk provides several enhancements around the usage of the blobstore service.

Getting blob information

Given a blob key, you can retrieve various details about the blob when it was uploaded:

```
BlobKey blob = ...

// retrieve an instance of BlobInfo
BlobInfo info = blob.info

// directly access the BlobInfo details from the key itself
String filename     = blob.filename
String contentType  = blob.contentType
Date creation       = blob.creation
long size           = blob.size
```

Serving blobs

With the blobstore service, you can stream the content of blobs back to the browser, directly on the response object:

```
BlobKey blob = ...

// serve the whole blob
blob.serve response

// serve a fragment of the blob
def range = new ByteRange(1000) // starting from 1000
blob.serve response, range

// serve a fragment of the blob using an int range
blob.serve response, 1000..2000
```

Reading the content of a Blob

Beyond the ability to serve blobs directly to the response output stream with **`blobstoreService.serve(blobKey, response)`** from your groovlet, there is the possibility of obtaining an **`InputStream`** to read the content of the blob. **Gaelyk** adds three convenient methods on **`BlobKey`** to easily deal with a raw input stream or with a reader, leveraging Groovy's own input stream and reader methods. The stream and reader are handled properly with regards to cleanly opening and closing those resources so that you don't have to take care of that aspect yourself.

```
BlobKey blobKey = ...

blobKey.withStream { InputStream stream ->
    // do something with the stream
}

// defaults to using UTF-8 as encoding for reading from the
   underlying stream
blobKey.withReader { Reader reader ->
    // do something with the reader
}

// specifying the encoding of your choice
blobKey.withReader("UTF-8") { Reader reader ->
    // do something with the reader
}
```

You can also fetch byte arrays for a given range:

```
BlobKey blob = ...
byte[] bytes

// using longs
bytes = blob.fetchData 1000, 2000

// using a Groovy int range
bytes = blob.fetchData 1000..2000

// using a ByteRange
def range = new ByteRange(1000, 2000) // or 1000..2000 as ByteRange
bytes = blob.fetchData range
```

Deleting a blob

Given a blob key, you can easily delete it thanks to the **`delete()`** method:

```
BlobKey blob = ...

blob.delete()
```

Iterating over and collecting `BlobInfos`

The blobstore service stores blobs that are identified by **`BlobKeys`**, and whose metadata are represented by **`BlobInfo`**. If you want to iterate over all the blobs from the blobstore, you can use the **`BlobInfoFactory`** and its **`queryBlobInfos()`** method, but **Gaelyk** simplifies that job with an **`each{}`** and a **`collect{}`** method right from the **`blobstore`** service:

```
blobstore.each { BlobInfo info -> out << info.filename }

def fileNames = blobstore.collect { BlobInfo info -> info.filename }
```

Example Blobstore service usage

In this section, we'll show you a full-blown example. First of all, let's create a form to submit a file to the blobstore, in a template named **`upload.gtpl`** at the root of your war:

```
<html>
<body>
    <h1>Please upload a text file</h1>
    <form action="${blobstore.createUploadUrl('/uploadBlob.groovy')}"
            method="post" enctype="multipart/form-data">
        <input type="file" name="myTextFile">
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

The form will be posted to a URL created by the blobstore service, that will then forward back to the URL you've provided when calling **`blobstore.createUploadUrl('/uploadBlob.groovy')`**

> ***Warning:*** *The URL to he groovlet to which the blobstore service will forward the uploaded blob details should be a direct path to the groovlet like* ***`/uploadBlob.groovy`****. For an unknown reason, you cannot use a URL defined through the URL routing system. This is not necessarily critical, in the sense that this URL is never deployed in the browser anyway.*

Now, create a groovlet named **`uploadBlob.groovy`** stored in **`/WEB-INF/groovy`** with the following content:

```
def blobs = blobstore.getUploadedBlobs(request)
def blob = blobs["myTextFile"]

response.status = 302

if (blob) {
    redirect "/success?key=${blob.keyString}"
} else {
    redirect "/failure"

}
```

In the groovlet, you retrieve all the blobs uploaded in the **`upload.gtpl`** page, and more particularly, the blob coming from the **`myTextFile`** input file element.

> ***Warning:*** *Google App Engine mandates that you explicitly specify a redirection status code (301, 302 or 303), and that you* ***do*** *redirect the user somewhere else, otherwise you'll get some runtime errors.*

We define some friendly URLs in the URL routing definitions for the upload form template, the success and failure pages:

```
get "/upload",  forward: "/upload.gtpl"
get "/success", forward: "/success.gtpl"
get "/failure", forward: "/failure.gtpl"
```

You then create a **`failure.gtpl`** page at the root of your war directory:

```
<html>
    <body>
        <h1>Failure</h1>
        <h2>Impossible to store or access the uploaded blob</h2>
    </body>
</html>
```

And a **`success.gtpl`** page at the root of your war directory, showing the blob details, and outputing the content of the blob (a text file in our case):

```
<% import com.google.appengine.api.blobstore.BlobKey %>
<html>
    <body>
        <h1>Success</h1>
        <% def blob = new BlobKey(params.key) %>

        <div>
            File name: ${blob.filename} <br/>
            Content type: ${blob.contentType}<br/>
            Creation date: ${blob.creation}<br/>
            Size: ${blob.size}
        </div>

        <h2>Content of the blob</h2>

        <div>
            <% blob.withReader { out << it.text } %>
        </div>
    </body>
</html>
```

Now that you're all set up, you can access **`http://localhost:8080/upload`**, submit a text file to upload, and click on the button. Google App Engine will store the blob and forward the blob information to your **`uploadBlob.groovy`** groovlet that will then redirect to the success page (or failure page in case something goes wrong).

File service

The File service API provides a convenient solution for accessing the blobstore, and particularly for programmatically adding blobs without having to go through the blobstore form-based upload facilities. **Gaelyk** adds a **`files`** variable in the binding of Groovlets and templates, which corresponds to the FileService instance.

Writing text content

Inspired by Groovy's own **`withWriter{}`** method, a new method is available on AppEngineFile that can be used as follows, to write text content through a writer:

```
// let's first create a new blob file through the regular
   FileService method
def file = files.createNewBlobFile("text/plain", "hello.txt")

file.withWriter { writer ->
    writer << "some content"
}
```

You can also specify three options to the **`withWriter{}`** method, in the form of named arguments:

- **encoding**: a string ("UTF-8" by default) defining the text encoding
- **locked**: a boolean (true by default) telling if we want an exclusive access to the file
- **finalize**: a boolean (true by default) to indicate if we want to finalize the file to prevent further appending

```
file.withWriter(encoding: "US-ASCII", locked: false, finalize:
   false) { writer ->
    writer << "some content"
}
```

Writing binary content

In a similar fashion, you can write to an output stream your binary content:

```
// let's first create a new blob file through the regular
   FileService method
def file = files.createNewBlobFile("text/plain", "hello.txt")

file.withOutputStream { stream ->
    stream << "Hello World".bytes
}
```

You can also specify two options to the **`withOutputStream{}`** method, in the form of named arguments:

- **locked**: a boolean (true by default) telling if we want an exclusive access to the file
- **finalize**: a boolean (true by default) to indicate if we want to finalize the file to prevent further appending

```
file.withOutputStream(locked: false, finalize: false) { writer ->
    writer << "Hello World".bytes
}
```

> ***Note:*** *To finalize a file in the blobstore, App Engine mandates the file needs to be locked. That's why by default* ***`locked`*** *and* ***`finalize`*** *are set to true by default. When you want to later be able to append again to the file, make sure to set* ***`finalize`*** *to false. And if you want to avoid others from concurrently writing to your file, it's better to set* ***`locked`*** *to false.*

Reading binary content

Gaelyk already provides reading capabilities from the blobstore support, as we've already seen, but the File service also supports reading from **AppEngineFile**s. To read from an **AppEngineFile** instance, you can use the **withInputStream{}** method, which takes an optional map of options, and a closure whose argument is a **BufferedInputStream**:

```
file.withInputStream { BufferedInputStream stream ->
    // read from the stream
}
```

You can also specify an option for locking the file (the file is locked by default):

```
file.withInputStream(locked: false) { BufferedInputStream stream ->
    // read from the stream
}
```

Reading text content

Similarily to reading from an input stream, you can also read from a **BufferedReader**, with the **withReader{}** method:

```
file.withReader { BufferedReader reader ->
    log.info reader.text
}
```

You can also specify an option for locking the file (the file is locked by default):

```
file.withReader(locked: false) { BufferedReader reader ->
    log.info reader.text
}
```

Miscelanous improvements

If you store a file path in the form of a string (for instance for storing its reference in the datastore), you need to get back an **AppEngineFile** from its string representation:

```
def path = someEntity.filePath
def file = files.fromPath(path)
```

If you have a **BlobKey**, you can retrieve the associated **AppEngineFile**:

```
def key = ... // some BlobKey
def file = key.file
```

You can retrieve the blob key associated with your file (for example when you want to access an **Image** instance:

```
def key = file.blobKey
def image = key.image
```

And if you want to delete a file without going through the blobstore service, you can do:

```
file.delete()
```

Namespace support

Google App Engine SDK allows you to create "multitenant"-aware applications, through the concept of namespace, that you can handle through the NamespaceManager class.

Gaelyk adds the variable **`namespace`** into the binding of your groovlets and templates. This **`namespace`** variable is simply the **`NamespaceManager`** class. **Gaelyk** adds a handy method for automating the pattern of setting a temporary namespace and restoring it to its previous value, thanks to the added **`of()`** method, taking a namespace name in the form of a string, and a closure to be executed when that namespace is active. This method can be used as follows:

```
// temporarily set a new namespace
namespace.of("customerA") {
    // use whatever service leveraging the namespace support
    // like the datastore or memcache
}
// once the closure is executed, the old namespace is restored
```

Images service enhancements

The images service and service factory wrapper

The Google App Engine SDK is providing two classes for handling images:

- **`ImageServiceFactory`** is used to retrieve the Images service, to create images (from blobs, byte arrays), and to make transformation operations.
- **`ImageService`** is used for applying transforms to images, create composite images, serve images, etc.

Very quickly, as you use the images handling capabilities of the API, you quickly end up jumping between the factory and the service class all the time. But thanks to **Gaelyk**, both **`ImagesServiceFactory`** and **`ImagesService`** are combined into one. So you can call any method on either of them on the same **`images`** instance available in your groovlets and templates.

```
// retrieve an image stored in the blobstore
def image = images.makeImageFromBlob(blob)

// apply a resize transform on the image to create a thumbnail
def thumbnail = images.applyTransform(images.makeResize(260, 260),
   image)

// serve the binary data of the image to the servlet output stream
sout << thumbnail.imageData
```

On the first line above, we created the image out of the blobstore using the images service, but there is also a more rapid shortcut for retrieving an image when given a blob key:

```
def blobKey = ...
def image = blobKey.image
```

In case you have a file or a byte array representing your image, you can also easily instanciate an **`Image`** with:

```
// from a byte array
byte[] byteArray = ...
def image = byteArray.image

// from a file directly
image = new File('/images/myimg.png').image
```

An image manipulation language

The images service permits the manipulation of images by applying various transforms, like resize, crop, flip (vertically or horizontally), rotate, and even an "I'm feeling lucky" transform! The **Gaelyk** image manipulation DSL allows to simplify the combination of such operations=

```
blobKey.image.transform {
    resize 100, 100
    crop 0.1, 0.1, 0.9, 0.9
    horizontal flip
    vertical flip
    rotate 90
    feeling lucky
}
```

The benefit of this approach is that transforms are combined within a single composite transform, which will be applied in one row to the original image, thus saving on CPU computation. But if you just need to make one transform, you can also call new methods on **`Image`** as follows:

```
def image = ...

def thumbnail   = image.resize(100, 100)
def cropped     = image.crop(0.1, 0.1, 0.9, 0.9)
def hmirror     = image.horizontalFlip()
def vmirror     = image.verticalFlip()
def rotated     = image.rotate(90)
def lucky       = image.imFeelingLucky()
```

Capabilities service support

Occasionally, Google App Engine will experience some reliability issues with its various services, or certain services will be down for scheduled maintenance. The Google App Engine SDK provides a service, the **`CapabilitiesService`**, to query the current status of the services. **Gaelyk** adds support for this service, by injecting it in the binding of your groovlets and templates, and by adding some syntax sugar to simplify its use.

```
import static com.google.appengine.api.capabilities.Capability.*
import static
   com.google.appengine.api.capabilities.CapabilityStatus.*

if (capabilities[DATASTORE] == ENABLED &&
   capabilities[DATASTORE_WRITE] == ENABLED) {
    // write something into the datastore
} else {
    // redirect the user to a page with a nice maintenance message
}
```

> ***Note:*** *Make sure to have a look at the capability-aware URL routing configuration.*

The services that can be queried are defined as static constants on **`Capability`** and currently are:

- BLOBSTORE
- DATASTORE
- DATASTORE_WRITE
- IMAGES
- MAIL

- MAIL
- MEMCACHE
- TASKQUEUE
- URL_FETCH
- XMPP

The different possible statuses are defined in the **`CapabilityStatus`** enum:

- ENABLED
- DISABLED
- SCHEDULED_MAINTENANCE
- UNKNOWN

> ***Tip:*** *Make sure to static import* ***`Capability`*** *and* ***`CapabilityStatus`*** *in order to keep your code as concise and readable as possible, like in the previous example, with:*
>
> ```
> import static
>    com.google.appengine.api.capabilities.Capability.*
> import static
>    com.google.appengine.api.capabilities.CapabilityStatus.*
> ```

Additionally, instead of comparing explicitely against a specific **`CapabilityStatus`**, **Gaelyk** provides a coercion of the status to a boolean (also called "Groovy Truth"). This allows you to write simpler conditionals:

```
import static com.google.appengine.api.capabilities.Capability.*
import static
   com.google.appengine.api.capabilities.CapabilityStatus.*

if (capabilities[DATASTORE] && capabilities[DATASTORE_WRITE]) {
    // write something into the datastore
} else {
    // redirect the user to a page with a nice maintenance message
}
```

> ***Note:*** *Only the* ***`ENABLED`*** *and* ***`SCHEDULED_MAINTENACE`*** *statuses are considered to be* ***`true`****, whereas all the other statuses are considered to be* ***`false`****.*

URLFetch Service improvements

Google App Engine offers the URLFetch Service to interact with remote servers, to post to or to fetch content out of external websites. Often, using the URL directly with Groovy's **`getBytes()`** or **`getText()`** methods is enough, and transparently uses the URLFetch Service under the hood. But sometimes, you need a bit more control of the requests you're making to remote servers, for example for setting specific headers, for posting custom payloads, making asynchronous requests, etc. **Gaelyk** 0.5 provides a convenient integration of the service with a groovier flavor.

> ***Note:*** *You may also want to have a look at HTTPBuilder's HttpURLClient for a richer HTTP client library that is compatible with Google App Engine.*

Gaelyk decorates the URL class with 5 new methods, for the 5 HTTP methods GET, POST, PUT, DELETE, HEAD which can take an optional map for customizing the call:

- **`url.get()`**
- **`url.post()`**
- **`url.put()`**

- ~~url.put()~~
- **url.delete()**
- **url.head()**

Those methods return an **HTTPResponse** or a **Future<HTTPResponse>** if the **async** option is set to true.

Let's start with a simple example, say, you want to get the **Gaelyk** home page content:

```
URL url = new URL('http://gaelyk.appspot.com')

def response = url.get()

assert response.responseCode == 200
assert response.text.contains('Gaelyk')
```

As you can see above, **Gaelyk** adds a **getText()** and **getText(String encoding)** method to HTTPResponse, so that it is easier to get textual content from remote servers — **HTTPResponse** only provided a **getContent()** method that returns a byte array.

If you wanted to make an asynchronous call, you could do:

```
def future = url.get(async: true)
def response = future.get()
```

Allowed options

Several options are allowed as arguments of the 5 methods.

- **allowTruncate**: a boolean (false by default), to explicit if we want an exception to be thrown if the reponse exceeds the 1MB quota limit
- **followRedirects**: a boolean (true by default), to specify if we want to allow the request to follow redirects
- **deadline**: a double (default to 10), the number of seconds to wait for a request to succeed
- **headers**: a map of headers
- **payload**: a byte array for the binary payload you want to post or put
- **params**: a map of query parameters
- **async**: a boolean (false by defauly), to specify you want to do an asynchronous call or not

To finish on the URLFetch Service support, we can have a look at another example using some of the options above:

```
URL googleSearch = "http://www.google.com/search".toURL()
HTTPResponse response = googleSearch.get(params: [q: 'Gaelyk'],
        headers: ['User-Agent': 'Mozilla/5.0 (Linux; X11)'])

assert response.statusCode == 200
assert response.text.contains('http://gaelyk.appspot.com')
assert response.headersMap.'Content-Type' == 'text/html;
   charset=utf-8'
```

> ***Note:*** ***response.statusCode*** *is a synonym of* ***response.responseCode****. And notice the convenient* ***response.headersMap*** *shortcut which returns a convenient* ***Map<String, String>*** *of headers instead of SDK's* ***response.headers****'s* ***List<HTTPHeader>****.*

Channel Service improvements

For your Comet-style applications, Google App Engine provides its Channel service. The API being very small, beyond adding the **`channel`** binding variable, **Gaelyk** only adds an additional shortcut method for sending message with the same **`send`** name as Jabber and Email support (for consistency), but without the need of creating an instance of **`ChannelMessage`**:

```
def clientId = "1234"
channel.createChannel(clientId)
channel.send clientId, "hello"
```

Backend service support

The backend service support is quite minimal, from a **Gaelyk** perspective, as only a **`backends`** (corresponding to a **`BackendService`** instance) and **`lifecylce`** (the **`LifecycleManager`**) variables have been added to the binding of Groovlets and templates.

In addition, a method for shutdown hooks was added that allows you to use a closure instead of a **`ShutdownHook`** instance:

```
lifecycle.shutdownHook = { /* shutting down logic */ }
```

Simple plugin system

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

This page is about creating new plugins. The list of existing plugins can be found in the Plugins section.

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 **Gaelyk** 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 Gaelyk plugin

A plugin is actually just some content you'll drop in your **`war/`** folder, at the root of your **Gaelyk** 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 Google App Engine 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 **Gaelyk** 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 **Gaelyk** 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 **Gaelyk** 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 (GAE, 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:

```
// 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* ***Gaelyk*** *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 **`gaelyk.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. Knowing that Google App Engine can load and unload apps depending on traffic, this is important to keep in mind as well.*

Using a plugin

If you recall, we mentioned the **`...`** script. This is a new script since **Gaelyk**

If you recall, we mentioned the **`plugins.groovy`** script. This is a new script since **Gaelyk** 0.4, 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* ***Gaelyk*** *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 **Gaelyk** **`war/`** folder, and reference that plugin, as explained in the previous section.

The best way how to share your plugin is by using the plugin catalogue. Fill the form and wait until the plugin is approved.

Running and deploying Gaelyk applications

> ***Note:*** *If you're using the template project as a base for your* ***Gaelyk*** *application, you should have a look at the section on the template project, which explains how you can use Gradle for your build, for running and deploying applications, and for testing your groovlets with Spock.*

Running your application locally

Google App Engine provides a local servlet container, powered by Jetty, which lets you run your applications locally. If you're using the **Gaelyk** template, when you're at the root of your project — and we assume you have installed the App Engine SDK on your machine — you can run your application with the following command-line:

```
dev_appserver.sh war
```

> ***Note:*** *Notice that there are some subtle differences between running locally and in the cloud. You'd better always check how your application works once deployed, as there may be some differences in behaviour between the two.*

Deploying your application in the cloud

Once you're at the root of your application, simply run the usual deployment command:

```
appcfg.sh update war
```