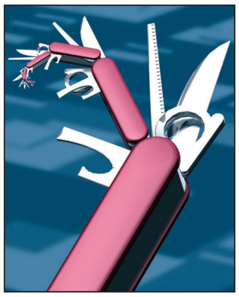
Getting Started with Google App Engine and Clojure



Aaron Bedra • *Relevance*

Over the past two years, Clojure (http://clojure.org) has made quite a splash in two areas: functional programming and the Java Virtual Machine (JVM). Clojure is a Lisp implementation on the JVM, offering its users significant elegance, simplicity, and power. Although Clojure is a fairly young language, it's extremely well thought out and mature, with a vibrant community of talented and friendly people using and contributing to it. If you're targeting the JVM as your platform, you're doing yourself a disservice by not considering Clojure as an option. That being said, beauty is in the eye of the beholder, and you must decide for yourself. So, let's take a look at Compojure, a Web framework built on the Clojure language, and see how to create and deploy a Compojure application on the Google App Engine platform (http://elhumidor.blogspot.com/2009/04/clojure-on-google-appengine.html).

Clojure

If you've ever programmed a Lisp dialect, most of Clojure's syntax will resonate with you quite well. Functions comprise a definition, a symbol (function name), an optional documentation string, a function signature, and a function body. The end result looks something like this:

```
(defn hello-world
  “Greets the world”
  [name]
  (println (str “Hello” name))
```

You can place several other options into a function, such as type hinting and metadata, but we won't cover those here. If you want to take a closer look at the language, Stuart Halloway's *Programming Clojure* is a great place to get started.[1]

Compojure

Clojure has a few Web frameworks available, such as Conjure (http://github.com/macourtney/Conjure), Cascade (http://github.com/hlship/cascade), and Compojure (http://github.com/weavejester/compojure). To date, Compojure has been the community favorite because it's a simple and concise Web framework that lets you build Web applications with an unmatched quickness. It's similar in functionality to the Sinatra (www.sinatrarb.com) framework from the Ruby World and is currently undergoing some pretty heavy changes. So, here, I've focused on the current stable version, 0.3.2. Version 0.4 is currently in beta but isn't quite ready for this publication. You'll be able to use Compojure 0.4 on App Engine, but the example code here won't quite do the trick.

Compojure's big win is simplicity; it lets you create a Web application in just a few lines of code. Compojure is also incredibly flexible, so you're empowered to build your application in a way that makes sense to you without having to fight the framework.

Google App Engine

The App Engine platform lets developers using Java and Python have an immediately available, free platform for hosting their Web applications. The service also lets users tap into Google's authentication services to identify and authorize users, taking some of the heavy lifting out of developing a Web application. Later, I'll show you how to spin up a Compojure-based Web application running on Google App Engine.

Getting Started

To get started, you'll need to download and install the Leiningen build tool (http://github.com/technomancy/leiningen), which uses Maven's dependency-management tools (http://maven.apache.org) and Ant's task-execution powers (http://ant.apache.org). You'll also need to download and have handy the Google App Engine Java toolkit.

```
(defproject example "1.0.0-SNAPSHOT"
  :description "IC example application"
  :dependencies [[org.clojure/clojure "1.2.0-master-SNAPSHOT"]
                 [org.clojure/clojure-contrib "1.2.0-SNAPSHOT"]
                 [compojure-gae "0.3.3"]]
  :compile-path "war/WEB-INF/classes"
  :library-path "war/WEB-INF/lib"
  :namespaces [example.core])
```

Figure 1. `project.clj` code. This example sets up the project dependencies so that Leiningen knows what to download and where to store the compiled code.

```
(ns example.core
  (:use [compojure.http servlet routes])
  (:gen-class :extends javax.servlet.http.HttpServlet))

(defroutes webservice
  (GET "/" "Hello App Engine"))
   (defservice webservice)
```

Figure 2. Program code. After Leiningen finishes fetching dependencies, open and modify `src/example/core.clj` to match the code in this figure.

Once you've installed Leiningen, it's time to set up the Leiningen project file and add the application's dependencies. Luckily, Leiningen has a nice feature that lets you generate a new project. Simply run `lein new example`, which will generate a project template that you can fill in. One such file will be a `project.clj` file stored in your application's root directory. Open it in your editor of choice and modify it to match the code in Figure 1.

Once you've completed your updates, run `lein deps`. This will fetch all the necessary libraries and their dependencies, with the exception of the App Engine jar files. After Leiningen is finished fetching dependencies, open `src/example/core.clj` and modify it to match the code in Figure 2.

This example pulls in Compojure and creates a servlet with a route to "/" that, when requested, will display the text "`Hello App Engine`". Both `defroutes` and `defservice` are macros inside the Compojure framework.

The last thing you need to do before you fire up your application and test it is create a `web.xml` and `appengine-web.xml` in `war/WEB-INF` using the code in Figure 3.

You now have enough to try out your example, but you must compile your application before you can boot it. You can do so by running `lein compile`. After this completes, you can start your application by running `dev_appserver.sh war` and opening your browser to `http://localhost:8080`. If you're having trouble running this command, make sure that you have the App Engine SDK's bin directory on your path.

Deploying Your App Engine Application

Now that you have a working application, it's time to deploy it to the App Engine service and test it out. At this point, you'll need to set up an account on App Engine. After you sign in to your account, you'll see a list of your applications. Click the Create an Application button. Choose an application identifier, give your application a title, and click Save. Once you've completed this step, you need to make one minor tweak before you're ready to deploy: open `appengine-web.xml` and modify the `application` tag to match the name of your App Engine application. When you're ready, simply run `appcfg.sh update war`. This starts the upload process and should leave you with a successfully deployed application.

Adding Some Style

With this structure, you might be wondering how to go about adding some real content and style to your application. You can do this many different ways, but Compojure has built-in HTML templating. Let's take a look at how to add basic style sheets and JavaScript files to an application.

First, create a `style.css` file in the `war/stylesheets` directory. Add a few basic style rules so you have something to reference. From here, open `example.core` and adjust it to match the code in Figure 4.

After making the changes, make sure to run `lein compile` and boot your application again. When you visit `http://localhost:8080` again, you should see your style sheet rules applied and working. If you want to see how this template language works and view a few more examples, you can visit http://github.com/weavejester/hiccup.

Authentication

An advantage to using App Engine is that you can tap into Google's authentication system, and doing so is quite easy. The first thing you'll need is the `appengine-api-1.0-sdk-1.3.3.jar` file. Copy this file into `war/WEB-INF/lib`. Next, create a new file called `user.clj` and add the code from Figure 5.

The example in Figure 5 taps

into the App Engine API and calls directly into some of the Java code that's provided in the toolkit. All the function calls that start with a period, such as `(.getCurrentUser (get-user-service))`, are actually Clojure's innovative Java interoperability at work. To take a closer look at how it works, visit http://clojure.org/java_interop. The code in Figure 5 isn't a complete authentication system, but it serves the basic purpose of logging in a user.

This code gives you all the basics you need to access the authentication system. The only thing left here is to wire up this code into the view. Open `core.clj` and update it to look like the code in Figure 6.

You now have what you need to test out the functionality. Once again, just run `lein compile` and boot your application. You should see the "Hello App Engine" text and a sign-in link. Clicking this link takes you to a fake authentication screen with a username already filled in. This is just the development environment stub for authentication, so clicking Log In will authenticate you without any passwords. Go ahead and deploy your application one more time and test it out with the real authentication system.

If you want to explore this further, you can find the example code for this article at http://github.com/abedra/ieee-example. The code includes examples for using Google's authentication system to identify users who access your application. To learn more about Compojure, you can join the google group at http://groups.google.com/group/compojure.

App Engine provides support for Java and Python. Although the examples provided by Google are written purely in Java, Clojure lets you tap into Java while opening up a world of simpler and more powerful abstractions. Of course, alternative JVM-based languages exist that offer advantages over using Java, but the elegance and simplicity of Clo-

```
# war/WEB-INF/web.xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"
     version="2.5">
     <servlet>
       <servlet-name>example</servlet-name>
       <servlet-class>example.core</servlet-class>
     </servlet>
     <servlet-mapping>
       <servlet-name>example</servlet-name>
       <url-pattern>/*</url-pattern>
     </servlet-mapping>
</web-app>

# war/WEB-INF/appengine-web.xml
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
  <application>example</application>
  <version>1</version>
  <static-files />
  <resource-files />
</appengine-web-app>
```

Figure 3. Program code. Use this code to create a `web.xml` and `appengine-web.xml` in `war/WEB-INF` to test your application.

```
(ns example.core
  (:use [compojure.http servlet routes]
        [compojure.html gen page-helpers])
  (:gen-class :extends javax.servlet.http.HttpServlet))

(defn index
  [request]
  (html
   (doctype :html4)
   [:head (include-css "/stylesheets/style.css")]
   [:body
    [:div {:id "content"} "Hello World"]]))

(defroutes webservice
  (GET "/" index))

(defservice webservice)
```

Figure 4. Adding style. This example uses the built in Compojure template language to build out the html content for the `"/"` route.

```
(ns example.user
  (:import com.google.appengine.api.users.
     UserServiceFactory))

(def user-service (atom nil))

(defn get-user-service
  "UserService for the current request."
  []
  (if @user-service
    @user-service
    (reset! user-service
      (UserServiceFacto-ry/getUserService))))

(defn get-user
  "If the user is not logged in will return
     nil."
  []
  (.getCurrentUser (get-user-service)))

(defn get-login-url
  ([dest]
   (.createLoginURL (get-user-service) dest))
  ([dest auth-domain]
   (.createLoginURL (get-user-service) dest
      auth-domain)))

(defn get-logout-url
  ([dest]
   (.createLogoutURL (get-user-service) dest))
  ([dest auth-domain]
   (.createLogoutURL (get-user-service) dest
      auth-domain)))

(defn is-logged-in
  []
  (.isUserLoggedIn (get-user-service)))

(defn login-box
  []
  (if (is-logged-in)
    (do  [:span {:class "login-text"}
       (get-user) " - "
          [:a {:href (get-logout-url "/")}
             "sign out"]])
    [:span {:class "login-text"}
     [:a {:href (get-login-url "/")}
       "sign in"]]))
```

Figure 5. Authentication. Use this code to tap into Google's authentication system.

```
(ns example.core
  (:use [compojure.http servlet routes]
        [compojure.html gen page-helpers]
        example.user)
  (:gen-class :extends javax.servlet.http.HttpServlet))

(defn render
  "The base layout for all pages"
  [body]
  (html
   (doctype :html4)
   [:head (include-css "/stylesheets/style.css")]
   [:body
    [:div {:class "container"}
     [:div {:id "login"}] (login-box)
     [:div {:id "content"} body]]]))

(defn index
  [request]
  (render "Hello App Engine"))

(defroutes webservice
  (GET "/" index))

(defservice webservice)
```

Figure 6. More authentication. Use this code to update `core.clj`.

jure combined with Compojure make for a great win on App Engine.

Reference

1. S. Halloway, *Programming Clojure, The Pragmatic Bookshelf*, May 2009; www.pragprog.com/titles/shcloj/programming-clojure.

Aaron Bedra is a principal at Relevance (http://thinkrelevance.com), where he works as a technical lead, speaker, and author. His research interests include enterprise systems integration using Clojure and JRuby. Bedra is a member of Clojure/core (http://clojure.com). He maintains the Ruby code coverage analysis tool, RCov, and is a contributor to many open source projects, including Clojure Contrib, Compojure, and Ruby on Rails. Contact him at aaron@thinkrelevance.com.