Heroku Integration

Heroku Plugin - Reference Documentation

Authors: Burt Beckwith

Version: 1.0

Table of Contents

1 Introduction to the Heroku Plugin

The Heroku plugin makes it easy to deploy Grails applications to Heroku.

The primary function of the plugin is that it is very Grails-aware and it will autoconfigure applicable cloud services using the Heroku environment. This includes the Hibernate DataSource and the Mongodb, Redis, RabbitMQ, and Memcached plugins.

1.1 History

History

- December 15, 2011
    - Initial 1.0 release

2 Usage

The first step is to install the plugin:

```
grails install-plugin heroku
```

Configuration

There is currently only one configuration option for the plugin, specified in Config.groovy:

Property	Default	Meaning
grails.plugin.heroku.datasource.disableTimeoutAutoconfiguration	`false`	disables auto-configuration of DataSource connection timeout checking if `true`

Creating an application

The general steps for creating a Grails application are (not necessarily in order):

- create and develop the application like any other application
- install the `heroku` plugin
- install the `mongodb`, `redis-gorm`, `rabbitmq`, and/or `memcached` plugins depending on which Heroku services you'll be using
- add the Heroku services you'll be using from the web interface (http://addons.heroku.com/) or with the `heroku addons:add` command
- create a local Git repo and commit your application files to it
- create the application on Heroku with the `heroku create` command
- deploy the application to Heroku's servers using `git push`

See the tutorial in the next section for a detailed walkthrough of the required and optional steps.

Using the Spring Security Core plugin

There are a few things to be aware of when using the spring-security-core plugin with Heroku. One is that as of this writing, there's no support at Heroku for session affinity or clustered sessions. So if you run more than one instance of your application, your authentications will tend to fail since after successfully authenticating you will most likely be redirected to another instance with a new HTTP session without the authentication. Even if this doesn't happen immediately, at some point you're likely to land on another instance and have to log in again. Until there is a workaround for this issue either use just a single node, or use a different security implementation.

Another is a more general problem with plugin dependencies that affects the spring-security-core plugin in Grails 2.0 applications. Due to changes in plugin resolution for 2.0, when the war is built the installed plugins' dependent plugins won't be resolved. So the webxml plugin that the spring-security-core plugin depends on won't be resolved and you'll see IllegalStateExceptions for all requests. This is the same issue that requires that you explicitly declare a dependency on the cloud-support plugin (see the tutorials for examples of this). The workaround is simple; just declare an explicit dependency for all plugins that would otherwise be transitively installed, e.g.

```
plugins {
   ...

compile ':spring-security-core:1.2.7.2'
    compile ':webxml:1.4.1'
    compile ':heroku:1.0'
    compile ':cloud-support:1.0.8'
}
```

If you're using 1.3.7 you can omit the webxml and cloud-support dependencies since they'll be resolved, but it doesn't hurt to have them there and will make upgrading to 2.0 easier.

A third issue has to do with SSL and channel security. Heroku uses F5 BIG-IP load balancers that affect how SSL is managed, so the default behavior of checking if a request is secure or insecure won't work. To address this just add a property in Config.groovy to enable checking for the request header value that their load balancers add:

```
grails.plugins.springsecurity.secureChannel.useHeaderCheckChannelSecurity =
true
```

This feature is only available in version 1.2.7.2 and higher of the spring-security-core plugin, so be sure to upgrade if you're using an older version.

3 Tutorials

To make things more clear, try one or both of these tutorials. The first is a basic tutorial that gets you started with a simple application that uses a PostgreSQL database and some domain classes. The advanced tutorial is more extensive and includes examples of using NoSQL and messaging.

3.1 Basic Tutorial

Here I assume you've got Grails 1.3.7 or 2.0.0 installed, along with Git and the Heroku command line client (as described in Getting Started with Java on Heroku/Cedar) and that you've authenticated to Heroku using the commandline client.

Do **not** create a pom.xml or Procfile.

Create the application

```
$ grails create-app herokutest
$ cd herokutest
```

Install the Heroku plugin

Register a dependency on the plugin (and also on the cloud-support plugin to be sure it's correctly resolved) in `grails-app/conf/BuildConfig.groovy` in the `plugins` section:

```
plugins {
   compile ':heroku:1.0'
   compile ':cloud-support:1.0.8'
}
```

Database configuration

You don't need to change anything in `DataSource.groovy` since the plugin reconfigures the settings when the application starts up. The plugin will set the `driverClassName` to 'org.postgresql.Driver' and the `dialect` to `org.hibernate.dialect.PostgreSQLDialect`, and change the `url`, `username`, and `password` to the values detected from the system properties for your PostgreSQL instance.

You'll need the JDBC driver for the PostgreSQL database, so add a dependency for it in `BuildConfig.groovy`. Add the `mavenCentral()` repository (and optionally `mavenLocal()`):

```
repositories {
   grailsPlugins()
   grailsHome()
   grailsCentral()

mavenLocal()
   mavenCentral()
}
```

and the jar dependency:

```
dependencies {
   runtime 'postgresql:postgresql:8.4-702.jdbc3'
}
```

Add a couple of domain classes so we can test the database:

```
$ grails create-domain-class database.Author
$ grails create-domain-class database.Book
```

Edit the classes so they look like these:

```
package database

class Author {
   String name

String toString() { name }

static hasMany = [books: Book]

static mapping = {
      cache true
   }
}
```

```
package database

class Book {
   String title

String toString() { title }

static belongsTo = [author: Author]

static mapping = {
      cache true
   }
}
```

Generate CRUD controllers and views for the domain classes:

```
$ grails generate-all database.Author
$ grails generate-all database.Book
```

Heroku and Git

Heroku uses Git to deploy your application, so initialize a Git repository:

```
$ git init
```

and you'll need a .gitignore file. In 2.0 you can run

```
$ grails integrate-with --git
```

but in 1.3.7 you need to create your own, e.g.

```
.settings
stacktrace.log
target
/web-app/plugins
/web-app/WEB-INF/classes
```

Check your application code into your Git repo:

```
$ git add .
$ git commit -m "initial commit"
```

Create the application at Heroku:

```
$ heroku create --stack cedar
```

This will generate a random name for your application, e.g. "evening-fog-8924". Yours will be different so where you see "evening-fog-8924" be sure to replace it with the name assigned to you. You can verify that your application is available by viewing its details at https://api.heroku.com/myapps.

Deploy

Deploying just involves pushing to the remote Git repository at Heroku:

```
$ git push heroku master
```

You'll see from the output that Heroku builds a war file from your project, including downloading dependencies and plugins. Once the push successfully completes you can view the log output with

```
$ heroku logs
```

and check status with

```
$ heroku ps
```

If the push fails you can fix the issues, commit, and try the push again.

If it worked, open the application in a browser by navigating to http://evening-fog-8924.herokuapp.com/ (replace "evening-fog-8924" with your actual application name).

Updates

When you update your code, run `git add` for the new and modified files, and commit. Then push again to the remote repo (`git push heroku master`) and your app will be stopped, rebuilt, and restarted.

3.2 Advanced Tutorial

Here I assume you've got Grails 1.3.7 or 2.0.0 installed, along with Git and the Heroku command line client (as described in Getting Started with Java on Heroku/Cedar) and that you've authenticated to Heroku using the commandline client.

Do **not** create a pom.xml or Procfile.

Create the application

```
$ grails create-app herokutest
$ cd herokutest
```

Install the Heroku plugin

Register a dependency on the plugin (and also on the cloud-support plugin to be sure it's correctly resolved) in `grails-app/conf/BuildConfig.groovy` in the `plugins` section:

```
plugins {
   compile ':heroku:1.0'
   compile ':cloud-support:1.0.8'
}
```

3.2.1 Database configuration

You don't need to change anything in `DataSource.groovy` since the plugin reconfigures the settings when the application starts up. The plugin will set the `driverClassName` to 'org.postgresql.Driver' and the `dialect` to `org.hibernate.dialect.PostgreSQLDialect`, and change the `url`, `username`, and `password` to the values detected from the system properties for your PostgreSQL instance.

You'll need the JDBC driver for the PostgreSQL database, so add a dependency for it in `BuildConfig.groovy`. Add the `mavenCentral()` repository (and optionally `mavenLocal()`):

```
repositories {
   grailsPlugins()
   grailsHome()
   grailsCentral()

mavenLocal()
   mavenCentral()
}
```

and the jar dependency:

```
dependencies {
   runtime 'postgresql:postgresql:8.4-702.jdbc3'
}
```

Add a couple of domain classes so we can test the database:

```
$ grails create-domain-class database.Author
$ grails create-domain-class database.Book
```

Edit the classes so they look like these:

```
package database

class Author {
   String name

String toString() { name }

static hasMany = [books: Book]

static mapping = {
      cache true
   }
}
```

```
package database

class Book {
   String title

String toString() { title }

static belongsTo = [author: Author]

static mapping = {
      cache true
   }
}
```

3.2.2 Heroku services

Now we'll add some other supported services. These are all optional, but each has a free version so there's no risk in testing them out.

Heroku uses Git to deploy your application, so initialize a Git repository:

```
$ git init
```

We need to create the application at Heroku so we can attach the services:

```
$ heroku create --stack cedar
```

This will generate a random name for your application, e.g. "evening-fog-8924". Yours will be different so where you see "evening-fog-8924" be sure to replace it with the name assigned to you. You can verify that your application is available by viewing its details at https://api.heroku.com/myapps.

3.2.3 Memcached

To use Memcached as your Hibernate 2nd-level cache, add a dependency for the `memcached` plugin in `BuildConfig.groovy`:

```
plugins {
   …
   compile ':memcached:1.0.2'
}
```

(use the latest version; find the value at the plugin page)

Also add the repositories for the plugin's dependencies:

```
repositories {
   …
   mavenRepo 'http://raykrueger.googlecode.com/svn/repository' // for
hibernate-memcached
   mavenRepo 'http://files.couchbase.com/maven2/'
}
```

and add an exclusion for ehcache since it's not needed:

```
inherits('global') {
   excludes 'ehcache'
}
```

Add the "memcache" service by running

```
$ heroku addons:add memcache:5mb
```

or you can choose a larger size if you want more capacity than the free version.

There are no configuration changes needed since the `heroku` plugin will configure things for you.

To see the new environment variables added for Memcached, run

```
$ heroku config
```

and the output should include something like

```
MEMCACHE_PASSWORD => your_password
MEMCACHE_SERVERS  => mc123.ec456.northscale.net
MEMCACHE_USERNAME => app12345%40heroku.com
```

3.2.4 MongoDB

If you want to try MongoDB you have two options, "MongoLab" or "MongoHQ". Both are supported, so either run

```
$ heroku addons:add mongolab:starter
```

or

```
$ heroku addons:add mongohq:free
```

To see the new environment variables added for the Mongo service you chose, run

```
$ heroku config
```

and the output should include something like

```
MONGOLAB_URI =>
mongodb://username:password@server.mongolab.com:27567/heroku_app12345
```

or

```
MONGOHQ_URL => mongodb://username:password@server.mongohq.com:10030/app12345
```

Also add a dependency for the `mongodb` plugin in `BuildConfig.groovy`:

```
plugins {
   …
   compile ':mongodb:1.0.0.RC3'
}
```

(use the latest version; find the value at the plugin page)

There are no configuration changes needed since the `heroku` plugin will configure things for you.

Create a domain class to test Mongo:

```
$ grails create-domain-class mongo.MongoThing
```

Edit the class so it looks like this:

```
package mongo

class MongoThing {
   String name
   Integer age
   Date dateCreated
   Date lastUpdated

static mapWith = 'mongo'
}
```

3.2.5 Redis

If you want to try Redis run

```
$ heroku addons:add redistogo:nano
```

To see the new environment variables added for the Redis service, run

```
$ heroku config
```

and the output should include something like

```
REDISTOGO_URL => redis://redistogo:6652dc88c@pike.redistogo.com:12345/
```

Also add a dependency for the `redis-gorm` plugin in `BuildConfig.groovy`:

```
plugins {
   …
   compile ':redis-gorm:1.0.0.M8'
}
```

(use the latest version; find the value at the plugin page)

There are no configuration changes needed since the `heroku` plugin will configure things for you.

Create a domain class to test Redis:

```
$ grails create-domain-class redis.RedisThing
```

Edit the class so it looks like this:

```
package redis

class RedisThing {
   String name
   Integer age
   Date dateCreated
   Date lastUpdated

static mapWith = 'redis'

static mapping = {
      name index: true
      age index: true
   }
}
```

3.2.6 RabbitMQ

If you want to try RabbitMQ ensure that your account has access to the private beta for RabbitMQ and run

```
$ heroku addons:add rabbitmq
```

To see the new environment variables added for the RabbitMQ service, run

```
$ heroku config
```

and the output should include something like

```
RABBITMQ_URL =>
amqp://username:password@server.rabbitmq.com:12345/virtualhost
```

Also add a dependency for the `rabbitmq` plugin in `BuildConfig.groovy`:

```
plugins {
   …
   compile ':rabbitmq:0.3.2'
}
```

(use the latest version; find the value at the plugin page)

Because of the way the RabbitMQ plugin configures itself, we do need to make some small configuration changes. We can use placeholder values for the username, password, and hostname since the `heroku` plugin will update those. While we're here, let's also configure a queue to work with ("herokuQueue"). We'll update the `production` block in Config.groovy with the Heroku values, and the `development` block with local values:

```
environments {
   production {
      rabbitmq {
         connectionfactory {
            username = 'placeholder'
            password = 'placeholder'
            hostname = 'placeholder'
            consumers = 5
         }
         queues = {
            herokuQueue()
         }
      }
   }
   development {
      rabbitmq {
         connectionfactory {
            username = 'guest'
            password = 'guest'
            hostname = 'localhost'
            consumers = 5
         }
         queues = {
            herokuQueue()
         }
      }
   }
}
```

Create a service to receive messages and peek at the most recent:

```
$ grails create-service rabbit.Message
```

Edit the class so it looks like this:

```
package rabbit

class MessageService {

static transactional = false

static rabbitQueue = 'herokuQueue'

// not thread-safe, just for demo
   List<String> mostRecentTextMessages = []

void handleMessage(String message) {
      try {
         log.info "Text message received at ${new Date()} $message"
         mostRecentTextMessages << message
         while (mostRecentTextMessages.size() > 10) {
            mostRecentTextMessages.remove 0
         }
      }
      catch (Throwable t) {
         // demo only - never catch Throwable!
         t.printStackTrace()
      }
   }
}
```

and a controller to send and view messages:

```
$ grails create-controller rabbit.Message
```

Edit the class so it looks like this:

```
package rabbit

class MessageController {

def messageService

def index = {}

def sendMessage = {
      rabbitSend 'herokuQueue', params.message
      flash.message = "Message sent: '$params.message'"
      redirect action: 'viewMessages'
   }

def viewMessages = {
      [messages: messageService.mostRecentTextMessages]
   }
}
```

Create `grails-app/views/message/index.gsp` to send messages:

```
<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta name="layout" content="main" />
<title>RabbitMQ Messaging</title>
</head>

<body>
   <div class="nav">
      <span class="menuButton">
         <a class="home" href="${createLink(uri: '/')}">Home</a>
      </span>
   </div>
   <div class="body">
      <h1>Send Message</h1>

<g:form action='sendMessage'>
         <div class="dialog">
            <table>
               <tbody>
                  <tr class="prop">
                     <td valign="top" class="name">
                       <label for="message">Message</label>
                     </td>
                     <td valign="top" class="value">
                        <g:textField name="message"/>
                     </td>
                  </tr>
               </tbody>
            </table>
         </div>
         <div class="buttons">
            <span class="button"><g:submitButton name="Send" /></span>
         </div>
      </g:form>
   </div>
</body>
</html>
```

and `grails-app/views/message/viewMessages.gsp` to view messages:

```
<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta name="layout" content="main" />
<title>RabbitMQ Messaging</title>
</head>

<body>

<div class="nav">
      <span class="menuButton">
         <a class="home" href="${createLink(uri: '/')}">Home</a>
      </span>
   </div>

<div class="body">

<h1>View Messages</h1>

<g:if test="${flash.message}">
      <div class="message">${flash.message}</div>
      </g:if>

<div class="list">
         <ul>
            <g:each in="${messages}" var="m">
            <li>${m.encodeAsHTML()}</li>
            </g:each>
         </ul>
      </div>

</div>
</body>
</html>
```

3.2.7 Tying it all together

We'll need a UI to view the domain classes, so run

```
$ grails generate-all "*"
```

to generate controllers and views for all of the domain classes.

Create an "info" controller that we can use to test the application's functionality:

```
$ grails create-controller info
```

Edit the class so it looks like this:

```
package herokutest

class InfoController {
   def index = {
      [env: System.getenv()]
   }
}
```

Optionally install the console plugin to provide a web-based console that runs arbitrary Groovy code, and also the dbconsole plugin (unless you're using 2.0 which has this feature built in). Add a reference to the console plugin in `BuildConfig.groovy`:

```
plugins {
   …
   compile ':console:1.0.1'
}
```

and if you're using 1.3.7 add a reference to the dbconsole plugin:

```
plugins {
   …
   compile ':dbconsole:1.1'
}
```

If you're using 2.0 you just need to enable it in the production environment since by default it's only enabled in development mode; enable it with the `grails.dbconsole.enabled` attribute in the production section of `Config.groovy`:

```
production {
   …
   grails.dbconsole.enabled = true
   …
}
```

If you're using 1.3.x add a dependency in `BuildConfig.groovy` for the `jquery` plugin so jQuery is available (it's automatically registered in 2.0 apps):

```
plugins {
   …
   compile ':jquery:1.7.1'
}
```

The GSP created here exposes a lot of information about your application and services, including passwords, a link to auto-login to a database console, and a web-based Groovy console that can run any arbitrary Groovy code. Be sure to guard access to your application with a security plugin, e.g. spring-security-core or Shiro

Create `grails-app/views/info/index.gsp`:

```
<html>

<head>
  <title>Heroku Grails Test</title>
  <meta name='layout' content='main' />
  <meta http-equiv='Content-Type' content='text/html; charset=UTF-8' />
  <style type="text/css" media="screen">
```

```
#nav {
    margin-top:20px;
    margin-left:30px;
    width:228px;
    float:left;

}
  .homePagePanel * {
    margin:0px;
  }
  .homePagePanel .panelBody ul {
    list-style-type:none;
    margin-bottom:10px;
  }
  .homePagePanel .panelBody h1 {
    text-transform:uppercase;
    font-size:1.1em;
    margin-bottom:10px;
  }
  .homePagePanel .panelBody {
    background: url(images/leftnav_midstretch.png) repeat-y top;
    margin:0px;
    padding:15px;
  }
  .homePagePanel .panelBtm {
    background: url(images/leftnav_btm.png) no-repeat top;
    height:20px;
    margin:0px;
  }

.homePagePanel .panelTop {
    background: url(images/leftnav_top.png) no-repeat top;
    height:11px;
    margin:0px;
  }
  h2 {
    margin-top:15px;
    margin-bottom:15px;
    font-size:1.2em;
  }
  #pageBody {
    margin-left:280px;
    margin-right:20px;
  }
  </style>
</head>

<body>
```

```
<body>
    <div id='nav'>
      <div class='homePagePanel'>
        <div class='panelTop'></div>
        <div class='panelBody'>
          <h1>Application Status</h1>
          <ul>
            <li>App version: <g:meta name='app.version'/></li>
            <li>Grails version: <g:meta name='app.grails.version'/></li>
            <li>Groovy version: ${GroovySystem.version}</li>
            <li>JVM version: ${System.getProperty('java.version')}</li>
            <li>Controllers:
${grailsApplication.controllerClasses.size()}</li>
            <li>Domains: ${grailsApplication.domainClasses.size()}</li>
            <li>Services: ${grailsApplication.serviceClasses.size()}</li>
            <li>Tag Libraries: ${grailsApplication.tagLibClasses.size()}</li>
          </ul>
          <h1>Installed Plugins</h1>
          <ul>
            <g:each var='plugin'
in='${applicationContext.pluginManager.allPlugins}'>
              <li>${plugin.name} - ${plugin.version}</li>
            </g:each>
          </ul>
        </div>
        <div class='panelBtm'></div>
      </div>
    </div>
    <div id='pageBody'>

<table>
        <thead>
          <tr><th>Name</th><th>Value</th></tr>
        </thead>
        <tbody>
          <tr>
            <td>DATABASE_URL</td>
            <td>${env.DATABASE_URL}</td>
          </tr>
          <tr>
            <td>RABBITMQ_URL</td>
            <td>${env.RABBITMQ_URL}</td>
          </tr>
          <tr>
            <td>REDISTOGO_URL</td>
            <td>${env.REDISTOGO_URL}</td>
          </tr>
          <tr>
            <td>MONGOHQ_URL</td>
            <td>${env.MONGOHQ_URL}</td>
          </tr>
          <tr>
            <td>MONGOLAB_URI</td>
            <td>${env.MONGOLAB_URI}</td>
          </tr>
          <tr>
            <td>MEMCACHE_SERVERS</td>
            <td>${env.MEMCACHE_SERVERS}</td>
          </tr>
          <tr>
            <td>MEMCACHE_USERNAME</td>
            <td>${env.MEMCACHE_USERNAME}</td>
          </tr>
          <tr>
            <td>MEMCACHE_PASSWORD</td>
            <td>${env.MEMCACHE_PASSWORD}</td>
          </tr>
        </tbody>
      </table>

<g:javascript library="jquery" plugin="jquery" />
```

```
<div id='controllerList' class='dialog'>
        <h2>Links:</h2>
        <ul>
          <li>Hibernate:
            <ul>
              <li class='controller'>
                <g:link controller='author'>Author Controller</g:link>
              </li>
              <li class='controller'>
                <g:link controller='book'>Book Controller</g:link>
              </li>
            </ul>
          </li>
          <li>Redis:
            <ul>
              <li class='controller'>
                <g:link controller='redisThing'>Redis Domain Class</g:link>
              </li>
            </ul>
          </li>
          <li>Mongo:
            <ul>
              <li class='controller'>
                <g:link controller='mongoThing'>Mongo Domain Class</g:link>
              </li>
            </ul>
          </li>
          <li>Rabbit:
            <ul>
              <li class='controller'>
                <g:link controller='message'>Send a message</g:link>
              </li>
              <li class='controller'>
                <g:link controller='message' action='viewMessages'>View
messages</g:link>
              </li>
            </ul>
          </li>
          <li>Admin:
            <ul>
              <li class='controller'>
                <g:link controller='console'>Console</g:link>
              </li>
              <li class='controller'>
                <g:link controller='dbconsole'>Database Console</g:link>
              </li>
              <li class='controller'>
                <h:dbconsoleLink>Database Console
(autologin)</h:dbconsoleLink>
              </li>
            </ul>
          </li>
        </ul>
      </div>
    </div>
  </body>
</html>
```

Logging

You should probably turn on debug logging in Config.groovy for the various plugins, e.g.

```
log4j = {
   error 'org.codehaus.groovy.grails',
         'org.springframework',
         'org.hibernate',
         'net.sf.ehcache.hibernate'
   debug 'grails.plugin.heroku',
         'grails.plugin.memcached',
         'grails.plugin.cloudsupport'
}
```

BootStrap

If you don't want to use the auto-login dbconsole link but still have the database console available, you can add some code to `BootStrap.groovy` to display the connect information. You can also print out environment variables and system properties while you're there:

```
import grails.plugin.heroku.PostgresqlServiceInfo

class BootStrap {

def init = { servletContext ->
      println "nSystem.getenv():"
      System.getenv().each { name, value ->
         println "System.getenv($name): $value"
      }
      println "n"
      println "nSystem.getProperties():"
      System.getProperties().each { name, value ->
         println "System.getProperty($name): $value"
      }
      println "n"

String DATABASE_URL = System.getenv('DATABASE_URL')
      if (DATABASE_URL) {
         try {
            PostgresqlServiceInfo info = new PostgresqlServiceInfo()
            println "nPostgreSQL service ($DATABASE_URL): url='$info.url', "
+
                    "user='$info.username', password='$info.password'n"
         }
         catch (e) {
            println "Error occurred parsing DATABASE_URL: $e.message"
         }
      }
   }
}
```

Run

```
$ heroku logs
```

after you've deployed your application to see this output.

Git

Heroku uses Git to deploy your application, so you'll need a .gitignore file. In 2.0 you can run

```
$ grails integrate-with --git
```

but in 1.3.7 you need to create your own, e.g.

```
.settings
stacktrace.log
target
/web-app/plugins
/web-app/WEB-INF/classes
```

Check your application code into your Git repo:

```
$ git add .
$ git commit -m "initial commit"
```

Deploy

Deploying just involves pushing to the remote Git repository at Heroku:

```
$ git push heroku master
```

You'll see from the output that Heroku builds a war file from your project, including downloading dependencies and plugins. Once the push successfully completes you can view the log output with

```
$ heroku logs
```

and check status with

```
$ heroku ps
```

If the push fails you can fix the issues, commit, and try the push again.

If it worked, open the application in a browser by navigating to http://evening-fog-8924.herokuapp.com/ (replace "evening-fog-8924" with your actual application name). Open the "info" controller page that has all of the links to the functionality of the application by navigating to http://evening-fog-8924.herokuapp.com/info

Updates

When you update your code, run `git add` for the new and modified files and commit. Then push again to the remote repo (`git push heroku master`) and your app will be stopped, rebuilt, and restarted.

4 Troubleshooting

Logging

Set the log level of the `heroku`, `cloud-support`, and/or `memcached` plugin classes to debug to view status messages while deploying if you're having issues:

```
log4j = {
   …
   debug 'grails.plugin.heroku',
         'grails.plugin.memcached',
         'grails.plugin.cloudsupport'
   …
}
```

console and dbconsole plugins

The console and dbconsole plugins are very helpful in diagnosing issues. The console plugin allows you to run arbitrary Groovy code from a web-based console (similar to the Grails/Groovy Swing-based console) and the dbconsole plugin exposes the H2 database's web-based database console (the H2 database console is available in Grails 2.0 by default, so you only need the plugin in pre-2.0 apps). The great thing about H2's database console is that it doesn't work for just H2 - it works for any JDBC database you have a driver for.

Add a reference to the console plugin in `BuildConfig.groovy`:

```
plugins {
   …
   compile ':console:1.0.1'
}
```

and if you're using 1.3.7 add a reference to the dbconsole plugin:

```
plugins {
   …
   compile ':dbconsole:1.1'
}
```

If you're using 2.0 you just need to enable it in the production environment since by default it's only enabled in development mode; enable it with the `grails.dbconsole.enabled` attribute in the production section of `Config.groovy`:

```
production {
   …
   grails.dbconsole.enabled = true
   …
}
```

These plugins are very dangerous if left exposed to the public. Be sure to guard them with security if you use them.

One issue you'll see is that it's tricky to know how to connect to your PostgreSQL database from the database console. You can use the console plugin to inspect the config settings, but it's more convenient to add this code to your application's `BootStrap.groovy`:

```
import grails.plugin.heroku.PostgresqlServiceInfo

class BootStrap {

def init = { servletContext ->
      String DATABASE_URL = System.getenv('DATABASE_URL')
      if (DATABASE_URL) {
         try {
            PostgresqlServiceInfo info = new PostgresqlServiceInfo()
            println "nPostgreSQL service ($DATABASE_URL): url='$info.url', "
+
                    "user='$info.username', password='$info.password'n"
         }
         catch (e) {
            println "Error occurred parsing DATABASE_URL: $e.message"
         }
      }
   }
}
```

One the application starts up you can view the output by running

```
$ heroku logs
```

You can also add a link to the database console that will automatically log you in using the plugin's taglib (replace the inner text with whatever you want to display as the link text):

```
<h:dbconsoleLink>Database Console (autologin)</h:dbconsoleLink>
```