heroku

Heroku for Java™ Workbook

Table of Contents

About the Heroku for Java Workbook

This workbook walks you through the steps of building Java apps that can run on Heroku.

Tip: You can find the latest version of this workbook, including all the source code, online at:

https://github.com/heroku/java-workbook

Prerequisites

Before you get started, you need to set up your development environment, as specified in the Prerequisites section of each tutorial. Lab machines are pre-configured with the tools that you'll need to use the workbook.

Terminology

We will start with some terminology that is helpful to understand.

PaaS

"Platform as a Service" is a general term for managed environments that run apps.

Heroku

A PaaS provider.

heroku

(Lower case "h") The command-line client for managing apps on Heroku.

git

A popular distributed version control system that is used to deploy apps to Heroku.

Heroku Add-on

The primary way Heroku can be extended. Add-ons are exposed as services that can be used from any app on Heroku.

Dyno

The isolated container that runs your web and other processes on Heroku.

Tutorial #1: Building a Web App

In this tutorial, you will create a web app and deploy it to Heroku. You will use a Maven archetype to create the app, which will include a web server. You'll first run the app locally, and then deploy it to Heroku using `git`.

Prerequisites

Working on your own computer:

You must:

- Install Maven
- Install `git`
- Create an SSH key and associate it with your Heroku account
- Install the `heroku` command-line client

Working on a lab computer:

None

Step 1: Create a Web App

1. Using the terminal or command line, navigate to the directory where you want to create the new project (this can be in your user's home directory). In the lab where multiple people use the same computer, you should create a new sub-directory for your code.

   Note: To launch a terminal window on a Mac, open the Terminal application in the **Applications** ➤ **Utilities** directory.

2. Run the following Maven command to generate a new project directory containing the basic web app structure, Maven dependencies and build definitions, and a Java class that will start an embedded Jetty web process:

```
mvn archetype:generate
-DarchetypeCatalog=http://maven.publicstaticvoidmain.net/archetype-catalog.xml
```

   You will be prompted for some properties for your project.

3. Select `1` for the `embedded-jetty-archetype`.
4. After the dependencies are downloaded, enter `foo` for the `groupId`.
5. Enter `helloheroku` for the `artifactId`.
6. Accept the defaults by pressing **Enter** for the version, package, and confirmation prompts.

A new project will be created in the `helloheroku` directory. This project contains everything needed for a Java web app.

- The `src/main/webapp` directory contains a default `index.html` page and the typical Java web app's `WEB-INF` directory.
- The `src/main/java/foo` directory contains a generated `Main.java` file that we will use later to start the web server process.
- The `helloheroku` directory contains a `pom.xml` file that contains the project configuration and dependencies for Maven.

Note: Mac and Linux use forward-slashes ("/") for file paths, and Windows uses back-slashes ("\"). The workbook lists variants of commands for Mac, Linux, and Windows. However, references to file paths and sample output only use forward-slashes.

Step 2: Test the App Locally

1. Compile and install the app into the local Maven repository by running:

```
cd helloheroku
mvn install
```

Maven creates startup scripts for the web app and installs a jar file in the local Maven repository. At the end of the output, you should see output similar to the following:

```
[INFO] Installing /home/jamesw/projects/helloheroku/target/helloheroku-1.0-SNAPSHOT.jar
 to /home/jamesw/.m2/repository/foo/helloheroku/1.0-SNAPSHOT/helloheroku-1.0-SNAPSHOT.jar
[INFO] Installing /home/jamesw/projects/helloheroku/pom.xml to
/home/jamesw/.m2/repository/foo/helloheroku/1.0-SNAPSHOT/helloheroku-1.0-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.980s
[INFO] Finished at: Fri Jul 22 07:30:10 MDT 2011
[INFO] Final Memory: 11M/167M
[INFO] ------------------------------------------------------------------------
```

2. Set the `REPO` environment variable to the location of the local Maven repository so the dependencies can be located when the web process is started:

On Mac or Linux:

```
export REPO=~/.m2/repository
```

On Windows:

```
set REPO=%UserProfile%\.m2\repository
```

3. Start the `webapp` process from the `helloheroku` directory:

On Mac or Linux:

```
sh target/bin/webapp
```

On Windows:

```
target\bin\webapp.bat
```

The `webapp` process has output similar to the following:

```
2011-07-22 07:32:37.439:INFO::jetty-7.4.4.v20110707
   2011-07-22 07:32:37.626:INFO::started
o.e.j.w.WebAppContext{/,file:/home/jamesw/projects/helloheroku/src/main/webapp/}
   2011-07-22 07:32:37.699:INFO::Started SelectChannelConnector@0.0.0.0:8080 STARTING
```

The Jetty web server process is now running.

4. Navigate to http://localhost:8080/ in your browser.

   You should see the following message:

   ```
   hello, world
   ```

5. Press `CTRL-C` to stop the process.

You are now ready to deploy this simple Java web app to Heroku.

Step 3: Deploy the Web App to Heroku

1. In the `helloheroku` project directory, create a new file named `Procfile` containing:

   ```
   web: sh target/bin/webapp
   ```

   Note: The file names, directories, and code are case sensitive. The `Procfile` file name must begin with an uppercase "P" character.

   Caution: Some text editors on Windows, such as Notepad, automatically append a `.txt` file extension to saved files. If that happens, you must remove the file extension.

   `Procfile` is a mechanism for declaring what commands are started when your dynos are run on the Heroku platform. In this case, we want Heroku to run the webapp startup script for web dynos.

2. Initialize a local git repository, add the files to it, and commit them:

   ```
   git init
   git add .
   git commit -m "initial commit for helloheroku"
   ```

   Note: On Windows, you can ignore the following message when running the "`git add .`" command:

   ```
   warning : LF will be replaced by CRLF in .gitignore
   ```

   The commit operation has output similar to the following:

   ```
   [master (root-commit) b914eee] initial commit
       7 files changed, 165 insertions(+), 0 deletions(-)
       create mode 100644 .gitignore
       create mode 100644 Procfile
       create mode 100644 README.md
       create mode 100644 pom.xml
       create mode 100644 src/main/java/foo/Main.java
       create mode 100644 src/main/webapp/WEB-INF/web.xml
       create mode 100644 src/main/webapp/index.html
   ```

3. Create a new app provisioning stack on Heroku by using the `heroku` command-line client:

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

Note: You must use the "cedar" stack when creating this new app because it's the only Heroku stack that supports Java.

The output looks similar to the following:

```
Creating empty-winter-343... done, stack is cedar
    http://empty-winter-343.herokuapp.com/ | git@heroku.com:empty-winter-343.git
    Git remote heroku added
```

Note: `empty-winter-343` is a randomly generated temporary name for the app. You can rename the app with any unique and valid name using the `heroku apps:rename` command.

The `create` command outputs the web URL and git URL for this app. Since you had already created a git repository for this app, the `heroku` client automatically added the heroku remote repository information to the git configuration.

4. Deploy the app to Heroku:

```
git push heroku master
```

This command instructs `git` to push the app to the master branch on the heroku remote repository. This automatically triggers a Maven build on Heroku. When the build finishes, the output ends with something like the following:

```
----->Discovering process types
     Procfile declares types -> web
-----> Compiled slug size is 17.0MB
-----> Launching... done, v6
http://empty-winter-343.herokuapp.com deployed to Heroku
To git@heroku.com:empty-winter-343.git
+ 3bcf805...a72152c master -> master (forced update)
```

5. Verify that the output contains the message: `Procfile declares types -> web`. If it doesn't, confirm that the `Procfile` is named correctly with no file extension and that it contains:

```
web: sh target/bin/webapp
```

If you fix `Procfile`, deploy the changes to Heroku:

```
git add Procfile
git commit -m "fixed Procfile"
git push heroku master
heroku scale web=1
```

6. Open the app in your browser using the generated app URL or by running:

```
heroku open
```

You should see `hello, world` on the web page.

Step 4: Scale the App on Heroku

By default, the app runs on one dyno. To add more dynos, use the `heroku scale` command.

1. Scale the app to two dynos:

```
heroku scale web=2
```

2. See a list of your processes:

```
heroku ps
```

Tip: This command is very useful as a troubleshooting tool. For example, if your web app is not accessible, use `heroku ps` to ensure that a web process is running. If it's not running, use `heroku scale web=1` to start the web app and use the `heroku logs` command to determine why there was a problem.

3. Scale back to one web dyno:

```
heroku scale web=1
```

Step 5: View App Logs on Heroku

You can see everything that your app outputs to the console (STDOUT and STDERR) by running the `heroku logs` command.

1. To see the logs, run:

```
heroku logs
```

2. To see log messages as they happen, use the "tail" mode:

```
heroku logs -t
```

3. Press `CTRL-C` to stop seeing a tail of the logs.

Step 6: Roll Back a Release on Heroku

Whenever you deploy code, change a config variable, or add or remove an add-on resource, Heroku creates a new release and restarts your app. You will learn more about add-ons in Tutorial #4: Using a Heroku Add-on on page 18.

You can list the history of releases, and use rollbacks to revert to prior releases to back out of bad deployments or config changes. This enables you to quickly revert to a known working state instead of creating a quick fix that might have other unforeseen effects.

1. To use the releases feature, install the `releases:basic` add-on.

```
heroku addons:add releases:basic
```

Note: If the output indicates that your app already has the add-on, you can ignore the message.

2. To try it out, change an environment variable for your app on Heroku:

   ```
   heroku config:add MYVAR=42
   ```

3. Now review your list of releases on Heroku:

   ```
   heroku releases
   ```

   You'll see a list of recent releases, including version number and the date of the release.

4. Roll back to the release before the `MYVAR` environment variable was set:

   ```
   heroku rollback
   ```

5. Verify that the `MYVAR` environment variable is no longer set:

   ```
   heroku config
   ```

Summary

In this tutorial, you created a web app and deployed it to Heroku. You learned how to push apps to Heroku using `git` and how the `Procfile` declares what commands are started when dynos are run. You also learned how to list and scale the number of dynos, view logs, and roll back releases.

Tutorial #2: Connecting to a Database

This tutorial extends the first tutorial by configuring your app to connect to a database. You will first test with a local PostgreSQL database and then use a shared PostgreSQL database on Heroku.

You could use JPA / Hibernate or other ORM frameworks to connect your Java app to the database, but in this tutorial we keep things very simple and use plain JDBC.

Prerequisites

Working on your own computer:

You must install and configure PostgreSQL.

Working on your own computer or on a lab computer:

You must complete Tutorial #1: Building a Web App on page 4.

Step 1: Configure the Web App to Use PostgreSQL

1. Navigate to the base directory of the app you created in Tutorial #1: Building a Web App.
2. Update the Maven dependencies to include the PostgreSQL JDBC driver by adding the following dependency inside the `<dependencies>` tag in the `pom.xml` file:

```
<dependency>
       <groupId>postgresql</groupId>
       <artifactId>postgresql</artifactId>
       <version>9.0-801.jdbc4</version>
</dependency>
```

Step 2: Create a Java DAO Class

Create a Data Access Object to read from and write to the database.

In the `src/main/java/foo` directory, create a new file named `TickDAO.java` with the following contents:

```
package foo;

import java.sql.*;

public class TickDAO {
   private static String dbUrl;

   static {
       dbUrl = System.getenv("DATABASE_URL");
       dbUrl = dbUrl.replaceAll(
           "postgres://(.*):(.*)@(.*)", "jdbc:postgresql://$3?user=$1&password=$2");
   }

   public int getTickCount() throws SQLException {
       return getTickcountFromDb();
   }

   public static int getScalarValue(String sql) throws SQLException {
       Connection dbConn = null;
       try {
           dbConn = DriverManager.getConnection(dbUrl);
           Statement stmt = dbConn.createStatement();
           ResultSet rs = stmt.executeQuery(sql);
           rs.next();
           System.out.println("read from database");
           return rs.getInt(1);
       } finally {
           if (dbConn != null) dbConn.close();
       }
   }

   private static void dbUpdate(String sql) throws SQLException {
       Connection dbConn = null;
       try {
           dbConn = DriverManager.getConnection(dbUrl);
           Statement stmt = dbConn.createStatement();
           stmt.executeUpdate(sql);
       } finally {
           if (dbConn != null) dbConn.close();
       }
   }

   private int getTickcountFromDb() throws SQLException {
       return getScalarValue("SELECT count(*) FROM ticks");
   }

   public static void createTable() throws SQLException {
       System.out.println("Creating ticks table.");
       dbUpdate("DROP TABLE IF EXISTS ticks");
       dbUpdate("CREATE TABLE ticks (tick timestamp)");
   }

   public void insertTick() throws SQLException {
       dbUpdate("INSERT INTO ticks VALUES (now())");
   }
}
```

Heroku uses environment variables for app configuration. The `DATABASE_URL` environment variable is automatically set for each dyno on Heroku. `TickDAO` expects a `DATABASE_URL` value in the following format:

```
postgres://[username]:[password]@[server]/[database-name]
```

`TickDAO` reads the `DATABASE_URL` environment variable and transforms it into the correct format for JDBC. When you run the app locally, you can set `DATABASE_URL` to point to your local database. You will do this in a later step.

Step 3: Create a JSP

Create a JSP file named `ticks.jsp` in the `src/main/webapp` directory. The file contents are:

```
<%@ page import="foo.*"%>
<html>
    <%
    TickDAO tickDAO = new TickDAO();
    tickDAO.insertTick();
    %>
    <%=tickDAO.getTickCount()%> Ticks
</html>
```

The `ticks.jsp` file uses the `TickDAO` to insert a new "tick" (row) into the database and then displays the number of ticks that have been inserted into the database.

Note: The `insertTick()` and `getTickCount()` calls are not wrapped in a database transaction so in this simple example it is possible that a user will not get the correct value if another user's "tick" is inserted between those calls. In a real-world app, database transactions should be used for these types of situations.

Step 4: Create a Java Schema Creation Class

1. In the `src/main/java/foo` directory, create a Java class named `SchemaCreator.java` that creates the database schema. The file contents are:

```
package foo;

import java.sql.SQLException;

public class SchemaCreator {
   public static void main(String[] args) throws SQLException {
       TickDAO tickDAO = new TickDAO();
       TickDAO.createTable();
   }
}
```

2. Navigate to the base directory of your app.
3. Update the `pom.xml` file to create a startup script for the SchemaCreator class by adding the following `<program>` block in the appassembler-maven-plugin's `<programs>` section:

```
<program>
   <mainClass>foo.SchemaCreator</mainClass>
   <name>schemaCreator</name>
</program>
```

Step 5: Test Database Access Locally

We will test on a local PostgreSQL database before deploying to Heroku.

1. Set the `DATABASE_URL` environment variable to point to your local PostgreSQL database so you can test locally before running on Heroku. We assume that you've set up PostgreSQL as defined in the appendix, with a database called `helloheroku` and a user `foo` with password `foo`.

   On Mac or Linux:

```
export DATABASE_URL=postgres://foo:foo@localhost/helloheroku
```

   On Windows:

```
set DATABASE_URL=postgres://foo:foo@localhost/helloheroku
```

2. Compile and install the app into the local Maven repository so that the webapp script can find it:

   ```
   mvn install
   ```

3. Create the database schema locally by running:

   On Mac or Linux:

   ```
   sh target/bin/schemaCreator
   ```

   On Windows:

   ```
   target\bin\schemaCreator.bat
   ```

4. Start the web app by running:

   On Mac or Linux:

   ```
   sh target/bin/webapp
   ```

   On Windows:

   ```
   target\bin\webapp.bat
   ```

   You should see output similar to the following:

   ```
   2011-07-20 18:14:07.342:INFO::jetty-7.4.4.v20110707
   2011-07-20 18:14:07.509:INFO::started
   o.e.j.w.WebAppContext{/,file:/home/jamesw/projects/java-workbook/tutorial-2/src/main/webapp/}
   2011-07-20 18:14:07.567:INFO::Started SelectChannelConnector@0.0.0.0:8080 STARTING
   ```

5. Verify that `ticks.jsp` works locally by navigating to http://localhost:8080/ticks.jsp in your browser.

   You should see `1 Ticks` on the web page. The tick count should increment by one every time you reload the page.

6. Press `CTRL-C` to stop the web app.

Step 6: Deploy the Web App to Heroku

1. Add the changes to git and commit:

   ```
   git add .
   git commit -m "added new DAO and JSP. Updated pom."
   ```

2. Deploy the new version of the app to Heroku:

   ```
   git push heroku master
   ```

3. We need to set up the database schema on the Heroku shared database. To do that, we will run the `schemaCreator` command just like we did locally, but this time on Heroku using the `heroku run` command:

   ```
   heroku run "sh target/bin/schemaCreator"
   ```

4. Navigate to `ticks.jsp` in your browser using the app's Heroku URL. The URL is similar to the following, but contains your unique app name instead of `empty-winter-343`:

   ```
   http://empty-winter-343.herokuapp.com/ticks.jsp
   ```

   You should see a web page that displays the number of ticks in the database. The tick count should increment by one every time you reload the page.

Summary

In this tutorial, you extended the app to connect to a database. The standard approach for configuring apps in Heroku is to use environment variables. You used the `DATABASE_URL` environment variable to set up the database connection.

You also learned how to use `heroku run` to execute one-off processes.

Tutorial #3: Running a Worker Process

A running app is a collection of processes. In previous tutorials, the app had a single process type–a web process–and you learned how to scale the number of web processes.

In this tutorial, you will build out your app to include a new process type: a worker.

Worker processes do "work" in the background and can only interact with a web process through other systems, such as a database or a messaging system. You can create multiple worker processes and each worker process can be run on multiple dynos.

The `Procfile` you created in Tutorial #1: Building a Web App enables you to define the process types in your app and how to start them.

In this tutorial, you will create a worker process, register it in your `Procfile`, and scale the number of worker processes on Heroku.

Prerequisites

Working on your own computer or on a lab computer:

You must complete Tutorial #2: Connecting to a Database on page 10.

Step 1: Create a Worker Java App

1. Create a worker Java app that adds a new "tick" to the database once a second. In the `src/main/java/foo` directory, create a new file named `Ticker.java` that contains:

```
package foo;

public class Ticker {

   public static void main(String[] args) {
       TickDAO tickDAO = new TickDAO();
       try {
           while (true) {
               tickDAO.insertTick();
               System.out.println("tick");
               Thread.sleep(1000);
           }
       }
       catch (Exception e) {
        // exception handling omitted for brevity
       }
   }
}
```

2. Navigate to the base directory of your app.
3. Update the `pom.xml` file to create a start script for the Ticker class by adding the following `<program>` block in the appassembler-maven-plugin's `<programs>` section:

```
<program>
   <mainClass>foo.Ticker</mainClass>
   <name>ticker</name>
</program>
```

This configures Maven to generate the `ticker` script used to start the worker process.

4. Save and close `pom.xml`.

Step 2: Test the App Locally

1. Test the `Ticker` class locally by compiling and installing the app into the local Maven repository:

```
mvn install
```

2. Run the `Ticker`:

   On Mac or Linux:

```
sh target/bin/ticker
```

   On Windows:

```
target\bin\ticker.bat
```

3. Verify that there are no errors. The process should output "tick" to the console once a second.
4. Press `CTRL-C` to stop the process.

Step 3: Deploy the App to Heroku

1. Navigate to the base directory of your app.
2. Update the `Procfile` to include the new worker process by adding the following line to the file:

```
tick: sh target/bin/ticker
```

3. Add the changes to git, commit, and deploy to Heroku:

```
git add .
git commit -m "added worker process"
git push heroku master
```

4. Start two dynos running the "tick" process:

```
heroku scale tick=2
```

5. Navigate to `ticks.jsp` in your browser using the app's Heroku URL. The URL is similar to the following, but contains your unique app name instead of `empty-winter-343`:

```
http://empty-winter-343.herokuapp.com/ticks.jsp
```

6. Refresh the page and notice that the number of ticks increments faster now that the `Ticker` processes are also adding new rows to the database. You can increase the tick rate further by adding more "tick" workers with the `heroku scale` command.

7. Turn off the "tick" worker processes so that they don't continue to use dyno hours:

```
heroku scale tick=0
```

Note: You are charged for dyno usage by the hour. Each app receives 750 free dyno hours per month, but it's a good idea to turn off worker processes if you're not using them.

8. Verify that the worker processes are no longer running:

```
heroku ps
```

Summary

In this tutorial, you learned about worker processes, which can do work in the background and communicate with your web processes through other systems, such as a database or a messaging system.

You also learned how to independently scale different process types on Heroku.

Tutorial #4: Using a Heroku Add-on

Heroku add-ons enable you to easily extend your app by using other cloud services.

In this tutorial, you will extend your app with distributed caching using Redis—an open source, NoSQL-style database. Redis is available through the Redis To Go Heroku add-on.

You will use Redis to cache the number of ticks to reduce the number of database requests. The cache will be set to expire after 30 seconds. This means that the tick count will only be updated every 30 seconds.

Prerequisites

Working on your own computer:

You must install and configure Redis.

Working on your own computer or on a lab computer:

You must complete Tutorial #3: Running a Worker Process on page 15.

Step 1: Configure the App to Use Redis for Caching

1. Navigate to the base directory of your app.
2. Update the `pom.xml` file to add the Jedis client library for Redis by adding the following to the `<dependencies>` section:

```
<dependency>
   <groupId>redis.clients</groupId>
   <artifactId>jedis</artifactId>
   <version>2.0.0</version>
</dependency>
```

3. Save and close `pom.xml`.

Step 2: Update the Java DAO Class to Use Redis

1. Open the `src/main/java/foo/TickDAO.java` file.
2. Add the following imports directly below the `import java.sql.*;` line:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.apache.commons.pool.impl.GenericObjectPool.Config;

import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.Protocol;
```

3. Add the following properties directly below the `public class TickDAO {` line:

```
   static JedisPool jedisPool;
   private static final String TICKCOUNT_KEY = "tickcount";
```

4. Add the following code below the `static {` line to set up a Redis connection pool:

```
        Pattern REDIS_URL_PATTERN = Pattern.compile(
            "^redis://([^:]*):([^@]*)@([^:]*):([^/]*)(/)?");
        Matcher matcher = REDIS_URL_PATTERN.matcher(System.getenv("REDISTOGO_URL"));
        matcher.matches();
        Config config = new Config();
        config.testOnBorrow = true;
        jedisPool = new JedisPool(config, matcher.group(3),
            Integer.parseInt(matcher.group(4)), Protocol.DEFAULT_TIMEOUT,
            matcher.group(2));
```

   You will set the `REDISTOGO_URL` environment variable in a later step. The environment variable is parsed using regular expressions when the connection pool is created.

5. Replace the `getTickCount()` method with the following code:

```
    public int getTickCount() throws SQLException {
        Jedis jedis = jedisPool.getResource();
        int tickcount = 0;
        String tickcountValue = jedis.get(TICKCOUNT_KEY);
        if (tickcountValue != null) {
            System.out.println("read from redis cache");
            tickcount = Integer.parseInt(tickcountValue);
        }
        else {
            tickcount = getTickcountFromDb();
            jedis.setex(TICKCOUNT_KEY, 30, String.valueOf(tickcount));
        }
        jedisPool.returnResource(jedis);

        return tickcount;
    }
```

   The `getTickCount()` method now checks for the tickcount in Redis. If it doesn't find it in the cache, it executes a database query and stores the value into Redis. The value is stored for 30 seconds.

6. Save and close `TickDAO.java`.

Step 3: Test the App Locally

1. Navigate to the base directory of your app.
2. Compile and install the app into the local Maven repository:

```
mvn install
```

3. Set the `REDISTOGO_URL` environment variable:

   On Mac or Linux:

```
export REDISTOGO_URL=redis://:@localhost:6379/
```

   On Windows:

```
set REDISTOGO_URL=redis://:@localhost:6379/
```

4. Run the `webapp` script to start the app locally:

   On Mac or Linux:

   ```
   sh target/bin/webapp
   ```

   On Windows:

   ```
   target\bin\webapp.bat
   ```

5. Navigate to http://localhost:8080/ticks.jsp in your browser.
6. Refresh the page and notice that the number of ticks is only updated every 30 seconds when the Redis cache expires.
7. Press `CTRL-C` to stop the web app.

Step 4: Deploy the App to Heroku

1. Add the free tier of the Redis To Go add-on to the `helloheroku` app:

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

   You should see output similar to the following:

   ```
   -----> Adding redistogo:nano to empty-winter-343... done, v20 (free)
   ```

   Note: If you're not in the lab, you have to verify your account before you can use the add-on. Verify your account by clicking:

   https://heroku.com/verify

2. Add the changes to git, commit, and deploy to Heroku:

   ```
   git add .
   git commit -m "added Redis support"
   git push heroku master
   ```

3. Navigate to `ticks.jsp` in your browser using the app's Heroku URL. The URL is similar to the following, but contains your unique app name instead of `empty-winter-343`:

   ```
   http://empty-winter-343.herokuapp.com/ticks.jsp
   ```

4. Verify that the number of ticks is only updated every 30 seconds by reloading the page a few times or by restarting the `tick` worker process.

Summary

In this tutorial, you learned how to take advantage of Heroku add-ons. In particular, you used the Redis To Go add-on to support caching in your app. You accessed the add-on by using an environment variable, similarly to how you extended your app to use a database in the previous tutorial.

Tutorial #5: Using Spring Roo

Spring Roo is a tool that makes building Java web apps very easy. For example, it can create CRUD objects, UI objects, and support general app development.

In this tutorial, you will create a simple Spring Roo web app, configure it, and then deploy it on Heroku. Instead of starting from scratch, this tutorial uses the Pet Clinic sample that comes with Spring Roo.

Prerequisites

Working on your own computer:

You must install Spring Roo.

Working on your own computer or on a lab computer:

You must complete Tutorial #2: Connecting to a Database on page 10.

Step 1: Create the Spring Roo Sample App

1. Using the terminal or command line, navigate to the sub-directory you created in Tutorial #1: Building a Web App for your code. This directory contains the `helloheroku` directory.
2. Create a new directory for the Spring Roo project and navigate to the directory:

```
mkdir petclinic
cd petclinic
```

3. Run the sample petclinic startup script.

```
roo script --file clinic.roo
```

Note: The path to `roo` was added to your `PATH` environment variable as part of the Spring Roo installation.

You now have a fully functional CRUD app!

Step 2: Configure the App's Runtime Environment

An app is "erosion resistant" if it continues to run without any maintenance from the developer. If you are running your own server, you typically need to manage operating system and security patches and many other system administration tasks.

Apps on Heroku are more erosion resistant if they explicitly state all their dependencies (including the app server) and can be quickly restarted. To do this, we will add the app server dependencies, the app dependencies, the app packaging, and the app startup process to your app’s configuration.

We will also switch the app from using an in-memory database to a PostgreSQL database, and from using Tomcat to using an embedded Jetty server.

To migrate from an external Tomcat to an embedded Jetty, we need to make a few changes to the Maven configuration and add a class that allows us to start up Jetty.

Finally, we will change the app packaging from a WAR file to a JAR file so that it can be easily executed in a standalone mode.

1. Navigate to the `petclinic` directory.
2. Open the `pom.xml` file for editing.
3. Add the Jetty server dependencies and PostgreSQL JDBC Driver to the `<dependencies>` section:

```
<dependency>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-webapp</artifactId>
    <version>7.4.4.v20110707</version>
</dependency>
<dependency>
    <groupId>org.mortbay.jetty</groupId>
    <artifactId>jsp-2.1-glassfish</artifactId>
    <version>2.1.v20100127</version>
</dependency>
<dependency>
    <groupId>postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.0-801.jdbc4</version>
</dependency>
```

4. Update the `<dependency>` sections for the following `artifactId` elements to change the `scope` value from `provided` to `compile` since Jetty doesn't include these components:

   - `servlet-api`
   - `org.springframework.roo.annotations`

   For example, change:

```
<dependency>
   <groupId>javax.servlet</groupId>
   <artifactId>servlet-api</artifactId>
   <version>2.5</version>
   <scope>provided</scope>
</dependency>
```

   to:

```
<dependency>
   <groupId>javax.servlet</groupId>
   <artifactId>servlet-api</artifactId>
   <version>2.5</version>
   <scope>compile</scope>
</dependency>
```

5. Add the `appassembler-maven-plugin` to the `plugins` section in the `build` tag. This enables the embedded Jetty process to be started from the command line.

```
<plugin>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>appassembler-maven-plugin</artifactId>
   <version>1.1.1</version>
   <executions>
       <execution>
           <phase>package</phase>
           <goals><goal>assemble</goal></goals>
           <configuration>
               <assembleDirectory>target</assembleDirectory>
```

```
                <generateRepository>false</generateRepository>
                <extraJvmArguments>-Xmx512m</extraJvmArguments>
                <programs>
                    <program>
                        <mainClass>Main</mainClass>
                        <name>webapp</name>
                    </program>
                </programs>
            </configuration>
        </execution>
    </executions>
</plugin>
```

6. Change the app packaging from `war` to `jar` by removing the following line:

```
<packaging>war</packaging>
```

7. Save and close the `pom.xml` file.

Step 3: Create a Java Class for Starting Jetty

Next, you will create a Java class to start an embedded Jetty instance.

In the `src/main/java` directory, create a new file named `Main.java` containing the following code:

```
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.webapp.WebAppContext;

public class Main
{
   public static void main(String[] args) throws Exception
   {
       String webappDirLocation = "src/main/webapp/";
       String webPort = System.getenv("PORT");

       if(webPort == null || webPort.isEmpty()) {
           webPort = "8080";
       }

       Server server = new Server(Integer.valueOf(webPort));
       WebAppContext root = new WebAppContext();

       root.setContextPath("/");
       root.setDescriptor(webappDirLocation+"/WEB-INF/web.xml");
       root.setResourceBase(webappDirLocation);
       root.setParentLoaderPriority(true);

       server.setHandler(root);
       server.start();
       server.join();
   }
}
```

Step 4: Configure the App to Use PostgreSQL

1. Modify the `src/main/resources/META-INF/spring/applicationContext.xml` file to use PostgreSQL by changing the following:

    ```
        <property name="url" value="${database.url}"/>
        <property name="username" value="${database.username}"/>
        <property name="password" value="${database.password}"/>
    ```

    to:

    ```
        <property name="url" value="#{systemEnvironment['DATABASE_URL'].replaceAll(
                 'postgres://(.*):(.*)@(.*)',
                 'jdbc:postgresql://$3?user=$1&amp;password=$2') }"/>
    ```

    The app will now look for database configuration information in the `DATABASE_URL` environment variable.
2. In the `src/main/resources/META-INF/spring/database.properties` file, change the `org.hsqldb.jdbcDriver` value to `org.postgresql.Driver` instead.
3. In the `src/main/resources/META-INF/persistence.xml` file, change the value of the `hibernate.dialect` to `org.hibernate.dialect.PostgreSQLDialect` instead.

Step 5: Test the App Locally

1. Set the `DATABASE_URL` environment variable to point to the local PostgreSQL database URL, and set the `REPO` environment variable:

    On Mac or Linux:

    ```
    export DATABASE_URL=postgres://foo:foo@localhost/helloheroku
    export REPO=~/.m2/repository
    ```

    On Windows:

    ```
    set DATABASE_URL=postgres://foo:foo@localhost/helloheroku
    set REPO=%UserProfile%\.m2\repository
    ```

2. Compile and install the app into the local Maven repository:

    ```
    mvn install
    ```

3. Start the app locally:

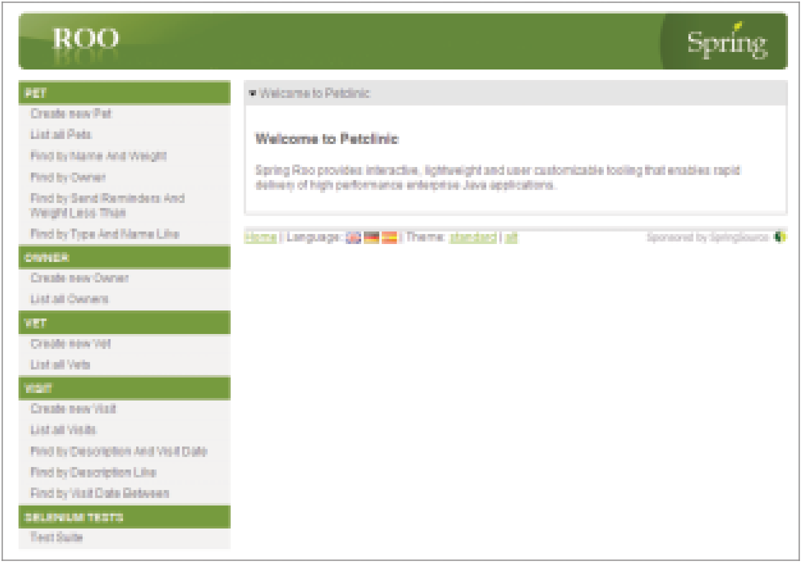
    On Mac or Linux:

    ```
    sh target/bin/webapp
    ```

    On Windows:

    ```
    target\bin\webapp.bat
    ```

4. Navigate to http://localhost:8080/ in your browser to try the app.



Welcome to Petclinic

Spring Roo provides interactive, lightweight and user customizable tooling that enables rapid delivery of high performance enterprise Java applications.

5. Press `CTRL-C` to stop the web app.

Step 6: Deploy the App to Heroku

1. Navigate to the `petclinic` directory.
2. Create a `Procfile` in this directory containing:

```
web:    sh target/bin/webapp
```

3. Create a git repository for this project, add the files to git, and commit them:

```
git init
git add pom.xml src Procfile
git commit -m "initial commit"
```

4. Create a new app on Heroku:

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

5. Deploy the app to Heroku:

```
git push heroku master
```

6. Open the app on Heroku:

```
heroku open
```

Summary

In this tutorial, you learned how to deploy a Spring Roo app to Heroku. You reconfigured the app to use an explicit process model by making it an executable JAR with an embedded Jetty server. This enables easy scaling on Heroku using the `Procfile`.

Next Steps

If you've completed this workbook, you now have a few Java apps deployed and running on Heroku. You also know how to scale your apps and extend them with Heroku add-ons.

To continue exploring:

- Visit http://devcenter.heroku.com/ to learn more about what you can do on the Heroku platform.
- Visit http://addons.heroku.com/ to learn about the add-ons that enable you to easily extend your app by using other cloud services.

APPENDIXES

Installing the `heroku` Command-Line Client

The `heroku` command-line client wraps the Heroku RESTful APIs and enables you to manage your apps on Heroku.

The client is written in Ruby. To install it, you first need to install Ruby, the Ruby package manager (RubyGems), and finally the client itself.

1. To install Ruby:
   - **On Mac**—Mac Snow Leopard and Lion ship with Ruby.
   - **On Ubuntu Linux**:

     ```
     sudo apt-get install ruby
     ```

   - **On Windows**—Use the RubyInstaller from:

     http://rubyinstaller.org/

2. To install RubyGems:

   a. Download the latest zip file from:

      http://rubygems.org/

   b. Uncompress the downloaded file and change directory to the expanded folder.

   c. Run the setup script in a terminal or command prompt:
      - **On Mac or Windows**:

        ```
        ruby setup.rb
        ```

      - **On Linux**:

        ```
        sudo ruby setup.rb
        sudo ln -s /usr/bin/gem1.8 /usr/bin/gem
        ```

3. To install the `heroku` client, run the following in a terminal or command prompt:
   - **On Mac or Windows**:

     ```
     gem install heroku
     ```

   - **On Linux**:

     ```
     sudo gem install heroku
     ```

4. Verify your installation by running the following command:

```
heroku version
```

5. Confirm that you see output similar to the following:

```
heroku-gem/2.3.6
```

Installing `git` and Setting up an SSH Key

Use `git` to upload your app to Heroku. It is a native app so follow the installation instructions for your operating system.

To upload your app to Heroku, you will need to create an SSH key (if you don't already have one) and associate it with your Heroku account.

1. Download the distribution of `git` for your operating system at:

   http://git-scm.com/download

2. Install `git` and create an SSH key by following the instructions in the link for your operating system. You only have to install `git` and create an SSH key. You don't need a GitHub account to complete this workbook.

   - **On Mac:** http://help.github.com/mac-set-up-git/
   - **On Windows:** http://help.github.com/win-set-up-git/

     Note: The instructions in this workbook assume that the directory containing `git.cmd` or `git.exe` is included in your `PATH` environment variable.

   - **On Linux:** http://help.github.com/linux-set-up-git/

3. If you haven't already created an account on http://heroku.com, create one now.
4. Log in to Heroku using a terminal or command line:

```
heroku auth:login
```

5. Associate your SSH key with your Heroku account:

```
heroku keys:add
```

Installing Maven

Apache Maven is a project management and build tool. A Maven project includes a project object model (POM) file that describes its dependencies. Maven can then manage these project dependencies when you are building and deploying your code.

1. Download the latest Maven 3 binary release at:

   http://maven.apache.org/download.html

2. Extract the archive and follow the installation instructions at:

   http://maven.apache.org/download.html#Installation

Installing PostgreSQL

PostgreSQL is an open source object-relational database system.

1. Download and install the PostgreSQL database by following instructions for your operating system:

   http://www.postgresql.org/download/

   Note: During installation, write down the installation directory that you choose as you'll have to navigate to the `bin` sub-directory later to create a database user.

2. Start the PostgreSQL server (if the installer didn't do so already).
3. Create a new user with superuser privileges, user name `foo`, and password `foo`:

   - **On Linux:**

     ```
     sudo -u postgres createuser -P foo
     ```

   - **On Mac:**

     ```
     createuser -P foo
     ```

   - **On Windows:**

     a. Using the terminal or command line, navigate to the `bin` sub-directory of your installation directory.
     b. Run the following command:

        ```
        createuser -U postgres -P foo
        ```

     c. Enter `foo` when you are prompted to `Enter password for new role` and to confirm the password.
     d. Enter `y` to make the new role a superuser.
     e. At the final `Password` prompt, enter the password for the postgres user.

4. Create a new database named `helloheroku`:

   a. Run the following command:

      ```
      createdb -U foo -W -h localhost helloheroku
      ```

   b. Enter `foo` when prompted for a password.

5. Test the connection to the database:

   ```
   psql -U foo -W -h localhost helloheroku
   ```

6. Verify that you see output similar to the following:

```
Password for user foo:
psql (9.0.4)

helloheroku=#
```

7. Type `\q` to exit the psql command line.

Installing Redis

Redis is an open source, advanced key-value store.

1. Install the Redis Server.
   - **On Ubuntu Linux**:

     ```
     sudo apt-get install redis-server
     ```

     The server starts by default.
   - **On Mac**: Follow the Redis installation instructions at:

     http://redis.io/download
   - **On Windows**: Download and install the Redis Server for Windows from:

     https://github.com/dmajkic/redis/downloads

     After uncompressing the files, start the server by executing:

     ```
     redis-server.exe
     ```

2. Verify it works by running `redis-cli`. You should see a command line but the prompt differs depending on your operating system:

```
redis 127.0.0.1:6379>
```

Installing Spring Roo

Spring Roo is a tool that makes building Java web apps very easy. For example, it can create CRUD objects, UI objects, and support general app development.

1. Download the latest Spring Roo GA release at:

   http://www.springsource.org/download
2. Follow the installation instructions at:

   http://static.springsource.org/spring-roo/reference/html/intro.html#intro-installation