Getting Started with Gradle on Heroku

Last updated 16 December 2019

Table of Contents

Introduction

This tutorial will have you deploying a Gradle app in minutes.

Hang on for a few more minutes to learn how it all works, so you can make the most out of Heroku.

The tutorial assumes that you have:

- a free Heroku account (https://signup.heroku.com/signup/dc)
- Java 8 installed

If you'd prefer to use Maven instead of Gradle, please see the Getting Started on Heroku with Java (https://devcenter.heroku.com/articles/getting-started-with-java) guide.

Set up

In this step you will install the Heroku Command Line Interface (CLI), formerly known at the Heroku Toolbelt. You will use the CLI to manage and scale your applications, to provision add-ons, to view the logs of your application as it runs on Heroku, as well as to help run your application locally.

Download and run the installer for your platform:

macOS

Download the installer (https://cli-assets.heroku.com/heroku.pkg)

Also available via Homebrew:

```
$ brew install heroku/brew/heroku
```

Windows

Download the appropriate installer for your Windows installation

64-bit installer (https://cli-assets.heroku.com/heroku-x64

32-bit installer (https://cli-assets.heroku.com/heroku-x86

Ubuntu 16+

Run the following from your terminal:

```
$ sudo snap install heroku --classic
```

Snap is available on other Linux OS's as well (https://snapcraft.io).

When installation completes, you can use the `heroku` command from your terminal.

Use the `heroku login` command to log in to the Heroku CLI:

```
$ heroku login
heroku: Press any key to open up the browser to login or q to exit
 ›   Warning: If browser does not open, visit
 ›   https://cli-auth.heroku.com/auth/browser/***
heroku: Waiting for login...
Logging in... done
Logged in as me@example.com
```

Authenticating is required to allow both the `heroku` and `git` commands to operate.

Note that if you're behind a firewall that requires use of a proxy to connect with external HTTP/HTTPS services, you can set the `HTTP_PROXY` or `HTTPS_PROXY` environment variables (https://devcenter.heroku.com/articles/using-the-cli#using-an-http-proxy) in your local development environment before running the `heroku` command.

Prepare the app

In this step, you will prepare a simple application that can be deployed.

Execute the following commands to clone the sample application:

```
$ git clone https://github.com/heroku/gradle-getting-started.git
$ cd gradle-getting-started
```

You now have a functioning Git repository that contains a simple application as well as a `build.gradle` file, which is used by the Gradle dependency manager.

Deploy the app

In this step you will deploy the app to Heroku.

Create an app on Heroku, which prepares Heroku to receive your source code:

```
$ heroku create
Creating app... done, arcane-inlet-19935
https://arcane-inlet-19935.herokuapp.com/ | https://git.heroku.com/arcane-inlet-19935.git
```

When you create an app, a Git remote (called `heroku`) is also created and associated with your local Git repository.

Heroku generates a random name (in this case `warm-eyrie-9006`) for your app, or you can pass a parameter to specify your own app name.

Now deploy your code:

```
$ git push heroku master
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Gradle app detected
remote: -----> Spring Boot detected
remote: -----> Installing JDK 1.8... done
...
```

The application is now deployed. Ensure that at least one instance of the app is running:

```
$ heroku ps:scale web=1
```

Now visit the app at the URL generated by its app name. As a handy shortcut, you can open the website as follows:

```
$ heroku open
```

View logs

Heroku treats logs as streams of time-ordered events aggregated from the output streams of all your app and Heroku components, providing a single channel for all of the events.

View information about your running app using one of the logging commands (https://devcenter.heroku.com/articles/logging), `heroku logs`:

```
$ heroku logs --tail
2019-02-07T18:17:07.989806+00:00 app[web.1]: 2019-02-07 18:17:07.989  INFO 4 --- [           main] o.s.b.a.w.s.WelcomePageHandlerMapping    : A
2019-02-07T18:17:08.543833+00:00 heroku[web.1]: State changed from starting to up
2019-02-07T18:17:08.538347+00:00 app[web.1]: 2019-02-07 18:17:08.538  INFO 4 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : T
2019-02-07T18:17:08.543461+00:00 app[web.1]: 2019-02-07 18:17:08.543  INFO 4 --- [           main] com.example.heroku.HerokuApplication     : S
2019-02-07T18:17:20.728559+00:00 app[web.1]: 2019-02-07 18:17:20.728  INFO 4 --- [io-48080-exec-3] o.a.c.c.C.[Tomcat].[localhost].[/]       : I
2019-02-07T18:17:20.728692+00:00 app[web.1]: 2019-02-07 18:17:20.728  INFO 4 --- [io-48080-exec-3] o.s.web.servlet.DispatcherServlet        : I
2019-02-07T18:17:20.742328+00:00 app[web.1]: 2019-02-07 18:17:20.742  INFO 4 --- [io-48080-exec-3] o.s.web.servlet.DispatcherServlet        : C
```

Visit your application in the browser again, and you'll see another log message generated. Press `Control+C` to stop streaming the logs.

Define a Procfile

Use a Procfile (https://devcenter.heroku.com/articles/procfile), a text file in the root directory of your application, to explicitly declare what command should be executed to start your app.

The `Procfile` in the example app you deployed looks like this:

```
web: java -jar build/libs/gradle-getting-started-1.0.jar
```

This declares a single process type, `web`, and the command needed to run it. The name `web` is important here. It declares that this process type will be attached to the HTTP routing (https://devcenter.heroku.com/articles/http-routing) stack of Heroku, and receive web traffic when deployed.

Procfiles can contain additional process types. For example, you might declare one for a background worker process that processes items off of a queue.

Scale the app

Right now, your app is running on a single web dyno (https://devcenter.heroku.com/articles/dynos). Think of a dyno as a lightweight container that runs the command specified in the `Procfile`.

You can check how many dynos are running using the `ps` command:

```
$ heroku ps
Free dyno hours quota remaining this month: 949h 54m (94%)
Free dyno usage for this app: 0h 0m (0%)
For more information on dyno sleeping and how to upgrade, see:
https://devcenter.heroku.com/articles/dyno-sleeping

=== web (Free): java -jar build/libs/gradle-getting-started-1.0.jar (1)
web.1: up 2019/02/13 11:10:03 -0600 (~ 13s ago)
```

By default, your app is deployed on a free dyno. Free dynos will sleep after a half hour of inactivity (if they don't receive any traffic). This causes a delay of a few seconds for the first request upon waking. Subsequent requests will perform normally. Free dynos also consume from a monthly, account-level quota of free dyno hours (https://devcenter.heroku.com/articles/free-dyno-hours) - as long as the quota is not exhausted, all free apps can continue to run.

To avoid dyno sleeping, you can upgrade to a hobby or professional dyno type as described in the Dyno Types (https://devcenter.heroku.com/articles/dyno-types) article. For example, if you migrate your app to a professional dyno, you can easily scale it by running a command telling Heroku to execute a specific number of dynos, each running your web process type.

For abuse prevention, scaling the application requires account verification (https://devcenter.heroku.com/articles/account-verification). If your account has not been verified, you will be directed to visit the verification site (https://heroku.com/verify).

Declare app dependencies

Heroku recognizes an app as a Gradle app by the existence of a `gradlew` or `build.gradle` file in the root directory.

The demo app you deployed already has a `build.gradle` (see it here (https://github.com/heroku/gradle-getting-started/blob/master/build.gradle)). Here's an excerpt:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.boot:spring-boot-starter-jdbc'
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    runtimeOnly 'org.postgresql:postgresql'
    runtimeOnly 'org.webjars:jquery:3.3.1-1'
    runtimeOnly 'org.webjars:jquery-ui:1.12.1'
    runtimeOnly 'org.webjars:bootstrap:4.1.3'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

The `build.gradle` file specifies dependencies that should be installed with your application. When an app is deployed, Heroku reads this file and installs the dependencies using the `./gradlew build` command.

Another file, `system.properties`, determines the version of Java to use. (Heroku supports many different versions (https://devcenter.heroku.com/articles/java-support#supported-java-versions)). The contents of this file, which is optional, are quite straightforward:

```
java.runtime.version=1.8
```

Run the Gradle `build` task in your local directory to install the dependencies, preparing your system for running the app locally. Note that this app requires Java 8, but that you can push your own apps using a different version of Java.

On Windows, run this comannd

```
> gradlew.bat build
```

On Mac and Linux run this command:

```
$ ./gradlew build
```

In either case, you'll see output like this:

```
BUILD SUCCESSFUL in 11s
5 actionable tasks: 5 executed
```

If you see an error such as `Unsupported major.minor version 52.0`, then Gradle is trying to use Java 7. Check that your `JAVA_HOME` environment variable is set correctly.

The Gradle process will copy the dependencies into a single JAR file in your application's `build/libs` directory. This process is called "vendoring", and it is done by default in a Spring app, such as the sample. But it can also be done manually as described in the Deploying Gradle Apps on Heroku (https://devcenter.heroku.com/articles/deploying-gradle-apps-on-heroku#verify-that-your-build-file-is-set-up-correctly) guide.

Once dependencies are installed, you will be ready to run your app locally.

Run the app locally

To run the app locally, first ensure that you've run the `gradlew build` task as described in the previous section. Then start your application using the `heroku local` command, which was installed as part of the Heroku CLI.

```
$ heroku local web
...
11:10:35 AM web.1 |  2019-02-13 11:10:35.008  INFO 62339 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing Executor
11:10:36 AM web.1 |  2019-02-13 11:10:36.713  INFO 62339 --- [           main] o.s.b.a.w.s.WelcomePageHandlerMapping    : Adding welcome page t
11:10:37 AM web.1 |  2019-02-13 11:10:37.175  INFO 62339 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on por
11:10:37 AM web.1 |  2019-02-13 11:10:37.179  INFO 62339 --- [           main] com.example.heroku.HerokuApplication     : Started HerokuApplicc
```

Just like Heroku, `heroku local` examines the `Procfile` to determine what to run. It also defines the port your app will bind to by setting the `PORT` environment variable, which is configured as `server.port` in the file `src/main/resources/application.properties`.

Your app will now be running at http://localhost:5000 (http://localhost:5000). Test that it's working with `curl` or a web browser, then `Ctrl+C` to exit.

`heroku local` doesn't just run your app - it also sets "config vars", something you'll encounter in a later tutorial.

Push local changes

In this step you'll learn how to propagate a local change to the application through to Heroku. As an example, you'll modify the application to add an additional dependency and the code to use it.

Modify `build.gradle` to include a dependency for `jscience` in the `dependencies` section like this:

In file `build.gradle`, on line 30 add:

```
compile "org.jscience:jscience:4.3.1"
```

Modify `src/main/java/com/example/heroku/HerokuApplication.java` so that it imports this library at the start.

In file `src/main/java/com/example/heroku/HerokuApplication.java`, on line 19 add:

```
import static javax.measure.unit.SI.KILOGRAM;
import javax.measure.quantity.Mass;
import org.jscience.physics.model.RelativisticModel;
import org.jscience.physics.amount.Amount;
```

Add the following `hello` method to `HerokuApplication.java`:

In file `src/main/java/com/example/heroku/HerokuApplication.java`, on line 59 add:

```
@RequestMapping("/hello")
String hello(Map<String, Object> model) {
    RelativisticModel.select();
    Amount<Mass> m = Amount.valueOf("12 GeV").to(KILOGRAM);
    model.put("science", "E=mc^2: 12 GeV = " + m.toString());
    return "hello";
}
```

In file `src/main/resources/templates/hello.html` write:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" th:replace="~{fragments/layout :: layout (~{::body},'hello')}">
<body>
  <div class="container">
    <p th:text="${science}"/>
  </div>
</body>
</html>
```

Here's the final source code (https://github.com/heroku/gradle-getting-started/blob/master/src/main/java/com/example/heroku/HerokuApplication.java) for `HerokuApplication.java` - yours should look similar. Here's a diff (https://github.com/heroku/gradle-getting-started/compare/localchanges) of all the local changes you should have made.

Now test locally:

```
$ ./gradlew build
...

BUILD SUCCESSFUL in 16s
5 actionable tasks: 5 executed
$ heroku local web
...
11:11:23 AM web.1 |  2019-02-13 11:11:23.455  INFO 63819 --- [           main] o.s.web.context.ContextLoader            : Root WebApplicationCc
11:11:24 AM web.1 |  2019-02-13 11:11:24.182  INFO 63819 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing Executor
11:11:24 AM web.1 |  2019-02-13 11:11:24.895  INFO 63819 --- [           main] o.s.b.a.w.s.WelcomePageHandlerMapping    : Adding welcome page t
11:11:25 AM web.1 |  2019-02-13 11:11:25.249  INFO 63819 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on por
```

Visiting your application on the `/hello` route at http://localhost:5000/hello (http://localhost:5000/hello), you should see some great scientific conversions displayed:

```
E=mc^2: 12 GeV = (2.139194076302506E-26 ± 1.4E-42) kg
```

Now deploy. Almost every deploy to Heroku follows this same pattern. First, add the modified files to the local Git repository:

```
$ git add .
```

Now commit the changes to the repository:

```
$ git commit -m "Demo"
```

Now deploy, just as you did previously:

```
$ git push heroku master
```

Finally, check that everything is working:

```
$ heroku open
```

Provision add-ons

Add-ons are third-party cloud services that provide out-of-the-box additional services for your application, from persistence through logging to monitoring and more.

By default, Heroku stores 1500 lines of logs from your application. However, it makes the full log stream available as a service - and several add-on providers have written logging services that provide things such as log persistence, search, and email and SMS alerts.

In this step you will provision one of these logging add-ons, Papertrail.

Provision the Papertrail (https://devcenter.heroku.com/articles/papertrail) logging add-on:

```
$ heroku addons:create papertrail
Creating papertrail on arcane-inlet-19935... free
Welcome to Papertrail. Questions and ideas are welcome (support@papertrailapp.com). Happy logging!
Created papertrail-objective-19491 as PAPERTRAIL_API_TOKEN
Use heroku addons:docs papertrail to view documentation
```

To help with abuse prevention, provisioning an add-on requires account verification (https://devcenter.heroku.com/articles/account-verification). If your account has not been verified, you will be directed to visit the verification site (https://heroku.com/verify).

The add-on is now deployed and configured for your application. You can list add-ons for your app like this:

```
$ heroku addons
```

To see this particular add-on in action, visit your application's Heroku URL a few times. Each visit will generate more log messages, which should now get routed to the Papertrail add-on. Visit the Papertrail console to see the log messages:

```
$ heroku addons:open papertrail
```

Your browser will open up a Papertrail web console, showing the latest log events. The interface lets you search and set up alerts:

Start a one-off dyno

You can run a command, typically scripts and applications that are part of your app, in a one-off dyno (https://devcenter.heroku.com/articles/one-off-dynos) using the `heroku run` command. It can also be used to launch a REPL process attached to your local terminal for experimenting in your app's environment, or code that you deployed with your application:

```
$ heroku run java -version
Running java -version on arcane-inlet-19935... up, run.6561 (Free)
openjdk version "1.8.0_201-heroku"
OpenJDK Runtime Environment (build 1.8.0_201-heroku-b09)
OpenJDK 64-Bit Server VM (build 25.201-b09, mixed mode)
```

If you receive an error, `Error connecting to process`, then you may need to configure your firewall (https://devcenter.heroku.com/articles/one-off-dynos#timeout-awaiting-process).

Don't forget to type `exit` to exit the shell and terminate the dyno.

Define config vars

Heroku lets you externalize configuration - storing data such as encryption keys or external resource addresses in config vars (https://devcenter.heroku.com/articles/config-vars).

At runtime, config vars are exposed as environment variables to the application. For example, modify `src/main/java/com/example/heroku/HerokuApplication.java` so that the method repeats grabs an energy value from the `ENERGY` environment variable:

In file `src/main/java/com/example/heroku/HerokuApplication.java`, on line 56 add:

```
@RequestMapping("/hello")
String hello(Map<String, Object> model) {
    RelativisticModel.select();
    String energy = System.getenv().get("ENERGY");
    if (energy == null) {
       energy = "12 GeV";
    }
    Amount<Mass> m = Amount.valueOf(energy).to(KILOGRAM);
    model.put("science", "E=mc^2: " + energy + " = "  + m.toString());
    return "hello";
}
```

Now compile the app again so that this change is integrated by running `./gradlew build` or `gradlew.bat build` respectively.

The `heroku local` command will automatically set up the environment based on the contents of the `.env` file in your local directory. In the top-level directory of your project there is already a `.env` file that has the following contents:

```
ENERGY=20 GeV
```

If you run the app with `heroku local web` and visit it at http://localhost:5000 (http://localhost:5000), you'll see the conversion value for 20 GeV.

To set the config var on Heroku, execute the following:

```
$ heroku config:set ENERGY="20 GeV"
Setting ENERGY and restarting arcane-inlet-19935... done, v7
ENERGY: 20 GeV
```

View the config vars that are set using `heroku config`:

```
$ heroku config
=== arcane-inlet-19935 Config Vars
ENERGY:               20 GeV
PAPERTRAIL_API_TOKEN: duMLEm2E5WHtf7mC163g
```

Deploy your changed application to Heroku to see this in action.

Use a database

The add-on marketplace (https://elements.heroku.com/addons/categories/data-stores) has a large number of data stores, from Redis and MongoDB providers, to Postgres and MySQL. In this step you will learn about the free Heroku Postgres add-on.

To begin, attach a new instance of the PostgreSQL add-on to your app by running this command:

```
$ heroku addons:create heroku-postgresql
Creating heroku-postgresql on arcane-inlet-19935... free
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pg:copy
Created postgresql-colorful-78005 as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation
```

You can find out a little more about the database provisioned for your app using the `addons` command in the CLI:

```
$ heroku addons

Add-on                                          Plan       Price  State
──────────────────────────────────────────────  ─────────  ─────  ───────
heroku-postgresql (postgresql-colorful-78005)  hobby-dev  free   created
 └─ as DATABASE

papertrail (papertrail-objective-19491)        choklad    free   created
 └─ as PAPERTRAIL

The table above shows add-ons and the attachments to the current app (arcane-inlet-19935) or other apps.
```

Listing the config vars for your app will display the URL that your app is using to connect to the database, `DATABASE_URL`:

```
$ heroku config
=== arcane-inlet-19935 Config Vars
DATABASE_URL:         postgres://pccigjsjmqxlvk:b3aedb1e099aa729d8037c0be6454e1028dc0c7c4dc16f8b366471486a8a7014@ec2-54-235-159-101.compute-1.c
ENERGY:               20 GeV
PAPERTRAIL_API_TOKEN: duMLEm2E5WHtf7mC163g
```

Heroku also provides a `pg` command that shows a lot more:

```
$ heroku pg
=== DATABASE_URL
Plan:                  Hobby-dev
Status:                Available
Connections:           10/20
PG Version:            10.6
Created:               2019-02-13 17:13 UTC
Data Size:             7.6 MB
Tables:                0
Rows:                  0/10000 (In compliance)
Fork/Follow:           Unsupported
Rollback:              Unsupported
Continuous Protection: Off
Add-on:                postgresql-colorful-78005
```

This indicates I have a hobby database (free), running Postgres 9.3.3, with a single row of data.

The example app you deployed already has database functionality, which you should be able to reach by visiting your app's URL and appending `/db`. For example, if your app was deployed to `https://wonderful-app-287.herokuapp.com/` then visit `https://wonderful-app-287.herokuapp.com/db`.

The code to access the database is straightforward. Here's the method to insert values into a table called `tick`:

```
  @Value("${spring.datasource.url}")
  private String dbUrl;

  @Autowired
  private DataSource dataSource;

  @RequestMapping("/db")
  String db(Map<String, Object> model) {
    try (Connection connection = dataSource.getConnection()) {
      Statement stmt = connection.createStatement();
      stmt.executeUpdate("CREATE TABLE IF NOT EXISTS ticks (tick timestamp)");
      stmt.executeUpdate("INSERT INTO ticks VALUES (now())");
      ResultSet rs = stmt.executeQuery("SELECT tick FROM ticks");

      ArrayList<String> output = new ArrayList<String>();
      while (rs.next()) {
        output.add("Read from DB: " + rs.getTimestamp("tick"));
      }

      model.put("records", output);
      return "db";
    } catch (Exception e) {
      model.put("message", e.getMessage());
      return "error";
    }
  }

  @Bean
  public DataSource dataSource() throws SQLException {
    if (dbUrl == null || dbUrl.isEmpty()) {
      return new HikariDataSource();
    } else {
      HikariConfig config = new HikariConfig();
      config.setJdbcUrl(dbUrl);
      return new HikariDataSource(config);
    }
  }
```

This ensures that when you access your app using the `/db` route, a new row will be added to the `tick` table, and all the rows will then be returned so that they can be rendered in the output.

The `DatabaseUrl.extract()` method, which is made available via the `heroku-jdbc` dependency in the `build.gradle`, retrieves the `DATABASE_URL` environment variable, set by the database add-on, and establishes a connection.

Deploy your change to Heroku by committing the changes to Git, and then running `git push heroku master`.

Now, when you access your app's `/db` route, you will see something like this:

```
Database Output
* Read from DB: 2014-08-08 14:48:25.155241
* Read from DB: 2014-08-08 14:51:32.287816
* Read from DB: 2014-08-08 14:51:52.667683
```

Assuming that you have Postgres installed locally (https://devcenter.heroku.com/articles/heroku-postgresql#local-setup), use the `heroku pg:psql` command to connect to the remote database and see all the rows:

```
$ heroku pg:psql
psql (10.1, server 9.6.10)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.

DATABASE=> SELECT * FROM ticks;
            tick
----------------------------
2018-03-01 20:53:27.148139
2018-03-01 20:53:29.288995
2018-03-01 20:53:29.957118
2018-03-01 21:07:28.880162
(4 rows)
=> \q
```

Read more about Heroku PostgreSQL (https://devcenter.heroku.com/articles/heroku-postgresql).

A similar technique can be used to install MongoDB or Redis add-ons (https://elements.heroku.com/addons/categories/data-stores).

Next steps

You now know how to deploy an app, change its configuration, view logs, scale, and attach add-ons.

Here's some recommended reading. The first, an article, will give you a more firm understanding of the basics. The last is a pointer to the main Java category here on Dev Center:

- Read How Heroku Works (https://devcenter.heroku.com/articles/how-heroku-works) for a technical overview of the concepts you'll encounter while writing, configuring, deploying and running applications.
- Read Deploying Gradle Apps on Heroku (https://devcenter.heroku.com/articles/deploying-gradle-apps-on-heroku) to understand how to take an existing Java app and deploy it to Heroku.
- Visit the Java category (https://devcenter.heroku.com/categories/java-support) to learn more about developing and deploying Java applications.