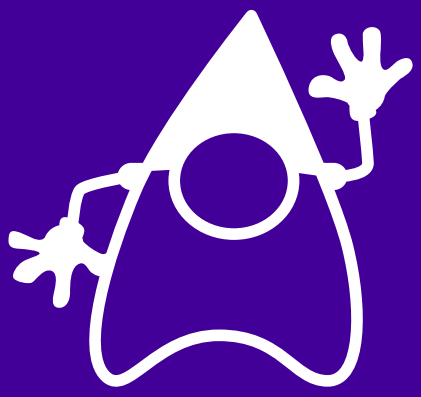


さあ Heroku をはじめよう

Java 編

HEROKU

1. はじめに

このチュートリアルでは、Java アプリケーションを短時間でリリースするためのステップを説明します。
全体的なステップをご理解いただき、Heroku の活用にお役立てください。
このチュートリアルは、次を前提としています。

- 無料の Heroku アカウントをお持ちであること(Herokuアカウントの登録方法は http://bit.ly/Heroku_SignUp_JP をご覧ください)
- Java 8 がローカルにインストールされていること
- Maven 3 がローカルにインストールされていること
  Maven の代わりに Gradle を使用する場合は、『Getting Started with Gradle on Heroku (Gradle を使用した Heroku の使用開始)』ガイドを参照してください。

2. 設定

このステップでは、Heroku Command Line Interface (CLI) (旧 Heroku Toolbelt) をインストールします。
この CLI を使用することで、ローカルでのアプリケーションの実行に加え、Heroku でのアプリケーションの管理および拡張、アドオンのプロビジョニング、アプリケーションの動作ログの表示が可能になります。

インストールが完了したら、コマンドシェルから heroku コマンドを使用できます。
Windows でコマンドシェルにアクセスするには、コマンドプロンプト (cmd.exe) または Powershell を起動します。
Heroku アカウントの作成時に使用したメールアドレスとパスワードを使用してログインします。

```
$ heroku login
Enter your Heroku credentials.
Email: java@example.com
Password:
```

heroku コマンドと git コマンドの両方を実行するためには、認証が必要です。
ローカルの開発環境がファイアウォールの内側にあり、外部の HTTP/HTTPS サービスに接続するためにプロキシを使用する必要がある場合は、`heroku` コマンドを実行する前に、ローカルの開発環境で環境変数 `HTTP_PROXY` または `HTTPS_PROXY` を設定します。

3. アプリケーションの準備

このステップでは、リリース用のシンプルなアプリケーションを準備します。
サンプルアプリケーションのクローンを作成して Heroku にリリースできるコードのローカル版を準備するには、ローカルのコマンドシェルまたはターミナルで次のコマンドを実行します。

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

これで、シンプルなアプリケーションおよび Java のパッケージマネージャー、Maven で使用される pom.xml ファイルが含まれるアクティブな git リポジトリが準備できました。

4. アプリケーションのリリース

このステップでは、アプリケーションを Heroku にリリースします。
Heroku にアプリケーションを作成し、Heroku がソースコードを取得できるように準備します。

```
$ heroku create
Creating warm-eyrie-9006... done, stack is cedar-14
http://warm-eyrie-9006.herokuapp.com/ | https://git.heroku.com/warm-eyrie-9006.git
Git remote heroku added
```

アプリケーションを作成すると、同時に git リモート（Heroku）も作成され、この git リモートはローカルの git リポジトリに関連付けられます。
作成したアプリケーションに対し、Heroku がランダムな名前を生成します（今回は warm-eyrie-9006）。
あるいは、パラメーターを渡してアプリケーション名を指定することもできます。
ここで、コードをリリースします。

```
$ git push heroku master
Initializing repository, done.
Counting objects: 68, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (19/19), done.
Writing objects: 100% (68/68), 7.07 KiB | 0 bytes/s, done.
Total 68 (delta 22), reused 65 (delta 22)

-----> Java app detected
-----> Installing OpenJDK 1.8... done
-----> Installing Maven 3.3.1... done
-----> Executing: mvn -B -DskipTests=true clean install

       [INFO] Scanning for projects...
       [INFO]
```

```
      [INFO] ------------------------------------------------------------------------
      [INFO] Building helloworld 1.0-SNAPSHOT
      [INFO] ------------------------------------------------------------------------
      ...
-----> Discovering process types
      Procfile declares types -> web

-----> Compressing... done, 62.7MB
-----> Launching... done, v7
      http://warm-eyrie-9006.herokuapp.com/ deployed to Heroku

To git@heroku.com:warm-eyrie-9006.git
 * [new branch]     master -> master
```

これで、アプリケーションがリリースされました。
アプリケーションのインスタンスを少なくとも 1 つ実行します。

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

アプリケーション名をもとに生成された URL でアプリケーションにアクセスします。
次の便利なショートカットを利用してアクセスすることもできます。

```
$ heroku open
```

5. ログの表示

Heroku では、すべてのアプリケーションおよび Heroku コンポーネントの出力ストリームからイベントを集約し、時系列に並べたイベントストリームとしてログを処理するため、すべてのイベントを単一のチャネルで確認できます。
`heroku logs --tail` を利用して実行中のアプリケーションでのログ記録コマンドからの情報を表示します。

```
$ heroku logs --tail
2015-07-06T08:44:45.008841+00:00 heroku[web.1]: Starting process with command
`java -jar target/helloworld.jar`
2015-07-06T08:44:47.941949+00:00 app[web.1]: [Thread-0] INFO spark.webserver.
SparkServer - == Spark has ignited ...
2015-07-06T08:44:47.949901+00:00 app[web.1]: [Thread-0] INFO org.eclipse.jetty.server.
Server - jetty-9.0.2.v20130417
2015-07-06T08:44:47.946093+00:00 app[web.1]: [Thread-0] INFO spark.webserver.
SparkServer - >> Listening on 0.0.0.0:6243
2015-07-06T08:44:48.280107+00:00 app[web.1]: [Thread-0] INFO org.eclipse.jetty.server.
ServerConnector - Started ServerConnector@42d76c7c{HTTP/1.1}{0.0.0.0:6243}
2015-07-06T08:44:48.703339+00:00 heroku[web.1]: State changed from starting to up
```

ブラウザーでアプリケーションに再度アクセスすると、別のログメッセージが生成されて表示されます。
`Control+C` を押して、ログのストリーミングを終了します。

6. Procfile の定義

アプリケーションのルートディレクトリにあるテキストファイル、Procfile を使用して、アプリケーションを起動するときに実行するコマンドを明示的に宣言します。
今回の例でリリースしたアプリケーションの `Procfile` は、次のとおりです。

```
web: java -jar target/helloworld.jar
```

これは 1 つのプロセスタイプ `web` と、その実行に必要なコマンドを宣言しています。ここで重要なのが `web` という名前です。このプロセスタイプは Heroku の HTTP ルーティングスタックにアタッチされ、リリース時に web トラフィックを受信することを宣言しています。
Procfile に追加のプロセスタイプを含めることもできます。たとえば、キューにあるアイテムをすべて処理するバックグラウンドワーカープロセスのプロセスタイプを宣言することもできます。

7. アプリケーションの拡張

ここまでの操作により、アプリケーションが1つの web dyno 上で動作しています。
dyno とは、`Procfile` に指定されたコマンドを実行する軽量のコンテナのようなものです。
動作している dyno の個数は、`ps` コマンドを使用して確認できます。

```
$ heroku ps
=== web (Free): java -jar target/helloworld.jar (1)
web.1: up 2016/07/07 09:06:41 +0100 (~ 4m ago)
```

デフォルトでは、アプリケーションは無料の dyno にリリースされます。無料の dyno は、30 分間操作がない場合（トラフィックの受信がまったくない場合）、スリープします。この状態になると、次のリクエスト時にスリープが解除されるまでに数秒の遅延が生じます。それに続くリクエストでは通常どおり動作します。
また、無料の dyno は、毎月アカウントレベルで割り当てられる無料 dyno 時間を消費します。
割り当てられた時間を使い切らない限り、すべての無料アプリケーションの動作が継続します。
dyno がスリープするのを防ぐため、『Dyno Types（dyno タイプ）』の記事に記載されている hobby またはプロフェッショナル用途の dyno タイプにアップグレードすることができます。たとえば、アプリケーションをプロ用途の dyno に移行すると、コマンドによって指定した数の dyno を起動し、それぞれで web プロセスタイプを実行するよう Heroku に命令することで、簡単に拡張できるようになります。
Heroku でアプリケーションを拡張または縮小するには、動作している dyno の個数を変更します。
web dyno の個数を 0 にするには次のようにします。

```
$ heroku ps:scale web=0
```

アプリケーションにアクセスし直すため、Web タブを更新するか、Web タブにアプリケーションを開く `heroku open` コマンドを実行すると、エラーメッセージが表示されます。
リクエストに応答できる web dyno が 1 つもなくなったためです。
再度拡張します。

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

不正使用を防止するため、アプリケーション内で有料の dyno を 2 つ以上に拡張するためにはアカウントの認証が必要です。

8. アプリケーションの依存関係の宣言

Heroku では、ルートディレクトリに`pom.xml`ファイルがあると、アプリケーションが Java であると認識されます。独自のアプリケーションの場合、`mvn archetype:create`コマンドを使用することでこのファイルを作成できます。
今回リリースしたデモアプリケーションには、すでに`pom.xml`(こちらを参照)が含まれています。
次に抜粋を示します。

```
<dependencies>
  <dependency>
   <groupId>com.sparkjava</groupId>
   <artifactId>spark-core</artifactId>
   <version>2.2</version>
  </dependency>
  <dependency>
   <groupId>com.sparkjava</groupId>
   <artifactId>spark-template-freemarker</artifactId>
   <version>2.0.0</version>
  </dependency>
  ...
 </dependencies>
```

この`pom.xml`ファイルが、アプリケーションとともにインストールする必要がある依存関係を指定します。アプリケーションのリリース時、Heroku はこのファイルを参照し、`mvn clean install`コマンドを用いて依存関係をインストールします。
別のファイル、`system.properties`は使用する Java のバージョンを指定します(Heroku は多数のバージョンをサポートします)。このファイルはオプションですが、その内容は非常に簡潔です。

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

アプリケーションをローカルで実行できるようシステムを準備するため、`mvn clean install`をローカルディレクトリで実行し、依存関係をインストールします。このアプリケーションには Java 8 が必要ですが、独自のアプリケーションの場合は他のバージョンの Java を使用してプッシュ可能です。

```
$ mvn clean install
INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building helloworld 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
...
Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/maven-archiver/
2.5/maven-archiver-2.5.pom
Downloaded: http://repo.maven.apache.org/maven2/org/apache/maven/maven-archiver/
2.5/maven-archiver-2.5.pom (5 KB at 14.9 KB/sec)
```

```
    ...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 5.584s
[INFO] Finished at: 2015-07-06T09:52:33+01:00
[INFO] Final Memory: 19M/222M
[INFO] ------------------------------------------------------------------------
```

Maven がインストールされていない場合や、[‘mvn’ は、内部コマンドまたは外部コマンドとして認識されていません] といったエラーが表示された場合、Windows では `mvnw clean install` を、Mac および Linux では `./mvnw clean install` を実行することで、代わりのラッパーコマンドを使用できます。
これにより、Maven のインストールと Maven コマンドの実行がどちらも行われます。
Maven プロセスが、依存関係を含めて JAR をコンパイルおよびビルドして、アプリケーションの `target` ディレクトリに配置します。このプロセスは、`pom.xml` ファイルにある次のプラグイン設定により実行されます。

```
<plugin>
 <artifactId>maven-assembly-plugin</artifactId>
 <version>2.3</version>
 <configuration>
 <descriptorRefs>
  <descriptorRef>jar-with-dependencies</descriptorRef>
 </descriptorRefs>
 <finalName>helloworld</finalName>
 ...
</plugin>
```

依存関係をインストールしたら、アプリケーションをローカルで実行する準備は完了です。

9. ローカルでのアプリケーションの実行

Heroku CLI の一部としてインストールされた heroku local コマンドを使用してアプリケーションをローカルで起動します（mvn clean install も実行済みであること）。

```
$ heroku local web
09:53:27 web.1  | started with pid 98562
09:53:28 web.1  | [Thread-0] INFO spark.webserver.SparkServer - == Spark has ignited ...
09:53:28 web.1  | [Thread-0] INFO spark.webserver.SparkServer - >> Listening on 0.0.0.0:5000
09:53:28 web.1  | [Thread-0] INFO org.eclipse.jetty.server.Server - jetty-9.0.2.v20130417
09:53:28 web.1  | [Thread-0] INFO org.eclipse.jetty.server.ServerConnector -
Started ServerConnector@1b4abf48{HTTP/1.1}{0.0.0.0:5000}
```

Heroku での場合と同様、heroku local が Procfile を検証し、何を実行すべきかを特定します。
Web ブラウザーで http://localhost:5000 を開きます。アプリケーションがローカルで実行されていることを確認します。
ローカルでのアプリケーションの実行を停止するには、ターミナルウィンドウに戻り、Ctrl+C を押して終了します。

10. ローカルで行った変更のプッシュ

このステップでは、アプリケーションに対してローカルで行った変更を Heroku に伝搬する方法を説明します。この例では、依存関係とそれを使用するためのコードを追加するという変更をアプリケーションに加えます。
pom.xml を変更して jscience の依存関係を追加します。<dependencies> セクションは次のようになります。

```
<dependencies>
 <dependency>
  <groupId>com.sparkjava</groupId>
  <artifactId>spark-core</artifactId>
  <version>2.2</version>
 </dependency>
 <dependency>
  <groupId>com.sparkjava</groupId>
  <artifactId>spark-template-freemarker</artifactId>
  <version>2.0.0</version>
 </dependency>
 <dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
```

```
<version>9.4-1201-jdbc4</version>
  </dependency>
  <dependency>
    <groupId>com.heroku.sdk</groupId>
    <artifactId>heroku-jdbc</artifactId>
    <version>0.1.0</version>
  </dependency>
  <dependency>
    <groupId>org.jscience</groupId>
    <artifactId>jscience</artifactId>
    <version>4.3.1</version>
  </dependency>
 </dependencies>
```

このライブラリが最初にインポートされるように、`src/main/java/Main.java`を変更して、次のインポートを追加します。

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

さらに、次のように get("/hello",...) メソッドを変更します。

```
get("/hello", (req, res) -> {
  RelativisticModel.select();
  Amount<Mass> m = Amount.valueOf("12 GeV").to(KILOGRAM);
  return "E=mc^2: 12 GeV = " + m.toString();
});
```

Main.java の完成したソースコードはこちらです。これと同様のソースコードが完成しているはずです。
これまでにローカルで行ったすべての変更の差分はこちらです。
ローカルでテストします。

```
$ mvn clean install
$ heroku local web
```

アプリケーションの /hello ルートに http://localhost:5000/hello でアクセスすると、次のように
科学単位の換算結果が表示されます。

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

リリースします。Heroku へのリリースは、ほぼ毎回同じパターンで行います。
まず、変更したファイルをローカルの git リポジトリに追加します。

```
$ git add .
```

次に、このリポジトリに変更をコミットします。

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

そして、先ほどと同様の方法でリリースします。

```
$ git push heroku master
```

最後に、すべてが正常に動作していることを確認します。

```
$ heroku open hello
```

11. アドオンのプロビジョニング

アドオンは、アプリケーションに標準の追加サービスを提供する、サードパーティのクラウドサービスです。
データの保持からログ記録、監視まで、さまざまな機能を提供します。
デフォルトでは、Heroku に保管できるアプリケーションログは 1,500 行です。ただし、完全なログストリームを利用できるサービスもあります。また、アドオンプロバイダー数社からログ記録サービスが提供されており、ログの保持、検索、メールおよび SMS でのアラートといった機能を利用できます。
このステップでは、こうしたログ記録アドオンの 1 つ、Papertrail をプロビジョニングします。
Papertrail ログ記録アドオンをプロビジョニングします。

```
$ heroku addons:create papertrail
Adding papertrail on warm-eyrie-9006... done, v8 (free)
Welcome to Papertrail. Questions and ideas are welcome (support@papertrailapp.com).
Happy logging!
Use `heroku addons:docs papertrail` to view documentation.
```

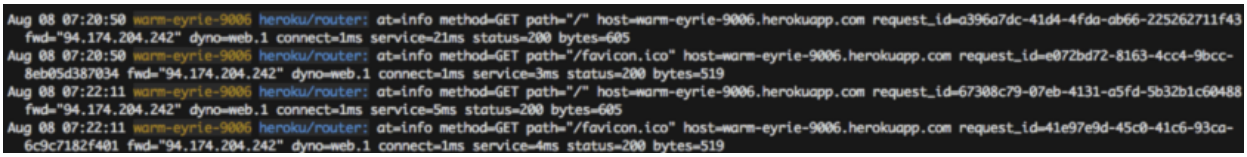
不正使用を防止するため、アドオンをプロビジョニングする際にはアカウントの認証が必要です。
アカウントの認証を行っていない場合は、認証サイトに転送されます。
これで、アドオンがアプリケーション用にリリースおよび設定されました。
次の方法で、アプリケーションのアドオンの一覧を確認できます。

```
$ heroku addons
```

この特定のアドオンの動作を確認するには、アプリケーションの Heroku URL に数回アクセスします。
アクセスするたびに新しいログメッセージが生成され、Papertrail アドオンに転送されるようになっています。ログメッセージを確認するには、Papertrail コンソールにアクセスします。

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

ブラウザーで Papertrail の Web コンソールが開き、最近のログイベントが表示されます。
このインターフェースから検索やアラートの設定を行えます。



12. one-off dyno の起動

アプリケーションの一部であるスクリプトやアプリケーションのコマンドは、`heroku run`コマンドを使用して one-off dyno で実行できます。このコマンドは、アプリケーション環境でのテストを目的として、ローカルのターミナルにアタッチされた REPL プロセスを起動したり、アプリケーションと合わせてリリースしたコードを実行する際にも使用できます。

```
$ heroku run bash
Running `bash` attached to terminal... up, run.9640
~ $ java -version
openjdk version "1.8.0_40-cedar14"
OpenJDK Runtime Environment (build 1.8.0_40-cedar14-b25)
OpenJDK 64-Bit Server VM (build 25.40-b25, mixed mode)
```

[`Error connecting to process`]というエラーが表示された場合、ファイアウォールの設定が必要である可能性があります。
必ず`exit`を入力してシェルを閉じ、dyno を終了します。

13. 設定変数の定義

Heroku では、暗号化鍵や外部リソースのアドレスといったデータを設定変数に格納することで、設定を外部化することができます。
アプリケーションの実行時、設定変数は環境変数としてアプリケーションに公開されます。たとえば、`src/main/java/Main.java`を変更して、環境変数`ENERGY`からエネルギー値を取得するようにします。

```
get("/hello", (req, res) -> {
    RelativisticModel.select();

    String energy = System.getenv().get("ENERGY");

    Amount<Mass> m = Amount.valueOf(energy).to(KILOGRAM);
    return "E=mc^2: " + energy + " = " + m.toString();
  });
```

この変更を反映するために、`mvn clean install`を実行してアプリケーションを再コンパイルします。`heroku local`を実行すると、ローカルディレクトリにある`.env`ファイルの内容に応じて環境が自動的に設定されます。プロジェクトの最上位ディレクトリには、次の内容の`.env`ファイルがすでにあります。

```
ENERGY=20 GeV
```

`heroku local`でアプリケーションを実行し、http://localhost:5000/hello にアクセスすると、20 GeV の換算値が表示されます。

```
$ heroku config:set ENERGY="20 GeV"
Setting config vars and restarting warm-eyrie-9006... done, v10
ENERGY: 20 GeV
```

`heroku config`を使用して、設定された設定変数を確認します。

```
$ heroku config
== warm-eyrie-9006 Config Vars
PAPERTRAIL_API_TOKEN: erdKhPeeeehIcdfY7ne
ENERGY: 20 GeV
...
```

変更したアプリケーションを Heroku にリリースし、動作を確認します。

14. データベースの使用

アドオンのマーケットプレイスには、Redis や MongoDB から Postgres や MySQL まで、プロバイダーによって数多くのデータストアが用意されています。このステップでは、Java アプリケーションのリリースで自動的にプロビジョニングされる、無料の Heroku Postgres アドオンについて説明します。
データベースはアドオンであるため、CLI の `addons` コマンドを使用すれば、アプリケーション用にプロビジョニングされたデータベースについて追加の情報を確認できます。

```
$ heroku addons
=== Resources for warm-eyrie-9006
Plan                   Name              Price
---------------------------  ------------------  -----
heroku-postgresql:hobby-dev  singing-aptly-6889  free
papertrail:choklad           gazing-nimbly-9108  free

=== Attachments for warm-eyrie-9006
Name        Add-on            Billing App
----------  ----------------  ----------------
DATABASE    singing-aptly-6889  warm-eyrie-9006
PAPERTRAIL  gazing-nimbly-9108  warm-eyrie-9006
```

アプリケーションの設定変数の一覧を表示すると、アプリケーションがデータベースへの接続に使用している `URL、DATABASE_URL` を確認できます。

```
$ heroku config
=== warm-eyrie-9006 Config Vars
DATABASE_URL:          postgres://qplhasewkhqyxp:YXDPSRus9MrU4HglCPzjhOevee@
ec2-54-204-47-58.compute-1.amazonaws.com:5432/dc9qsdnghia6v1
...
```

Heroku では、より詳細な情報を表示できる pg コマンドも提供されています。

```
$ heroku pg
== HEROKU_POSTGRESQL_BLUE_URL (DATABASE_URL)
Plan:       Hobby-dev
Status:     Available
Connections: 0
PG Version:  9.3.3
Created:    2014-08-08 13:54 UTC
Data Size:  6.5 MB
Tables:     0
Rows:       0/10000 (In compliance)
Fork/Follow: Unsupported
Rollback:   Unsupported
```

これは、Postgres 9.3.3 を実行し、hobby データベース（無料）に 1 行のデータが格納されていることを示しています。
今回の例でリリースしたアプリケーションには、すでにデータベース機能があり、この機能には、アプリケーションの URL に /db を追加することでアクセスできます。たとえば、アプリケーションのリリース先が https://wonderful-app-287.herokuapp.com/ の場合、https://wonderful-app-287.herokuapp.com/db でアクセスできます。
データベースにアクセスするためのコードはシンプルです。tick というテーブルに値を挿入するメソッドを次に示します。

```
HikariConfig config = new  HikariConfig();
config.setJdbcUrl(System.getenv("JDBC_DATABASE_URL"));
final HikariDataSource dataSource = (config.getJdbcUrl() != null) ?
 new HikariDataSource(config) : new HikariDataSource();

get("/db", (req, res) -> {
 Map<String, Object> attributes = new HashMap<>();
 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"));
  }
```

```
    attributes.put("results", output);
    return new ModelAndView(attributes, "db.ftl");
  } catch (Exception e) {
    attributes.put("message", "There was an error: " + e);
    return new ModelAndView(attributes, "error.ftl");
  }
}, new FreeMarkerEngine());
```

これで、`/db`ルートを使用してアプリケーションにアクセスすると、`tick`テーブルに新しい行が追加されるようになりました。さらに、すべての行が返却されて、画面に表示されます。
データベース接続プールの HikariCP は、データベースのアドオンにより設定された環境変数`JDBC_DATABASE_URL`を使用して初期化され、このデータベースへの接続プールを確立します。
変更を Heroku にリリースします。まずは変更を git にコミットし、`git push heroku master`を実行します。
これで、アプリケーションの`/db`ルートにアクセスすると、次のように表示されるようになります。

```
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
```

Postgres がローカルにインストールされていることを前提として、`heroku pg:psql`コマンドを使用してリモートのデータベースに接続し、すべての行を表示します。

```
$ heroku pg:psql
heroku pg:psql
psql (9.3.2, server 9.3.3)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.
=> SELECT * FROM ticks;
          tick
----------------------------
 2014-08-08 14:48:25.155241
 2014-08-08 14:51:32.287816
 2014-08-08 14:51:52.667683
 2014-08-08 14:51:53.1871
 2014-08-08 14:51:54.75061
 2014-08-08 14:51:55.161848
 2014-08-08 14:51:56.197663
 2014-08-08 14:51:56.851729
(8 rows)
=> \q
```

Heroku PostgreSQL の詳細をご覧いただけます。
MongoDB および Redis アドオンのインストールにも、同様の技法を使用できます。

15. 次のステップ

ここまでで、アプリケーションのリリース、設定の変更、ログの表示、拡張、アドオンの追加の方法をお伝えしました。
ここで、おすすめの補足資料をご案内します。1 つ目は基礎をより一層固めるのに役立つ記事で、2 つ目はここ Dev Center のメインカテゴリ、Java のページです。

- 『How Heroku Works（Heroku の仕組み）』では、アプリケーションの記述、設定、リリース、実行時に直面する概念の技術的な概要を説明しています。
- 『Deploying Java Apps on Heroku（Heroku での Java アプリケーションのリリース）』では、既存の Java アプリケーションを Heroku 用に移植して Heroku にリリースする方法を説明しています。
- Java カテゴリでは、Java アプリケーションの開発とリリースについて詳しく説明しています。