**Introduction**

Deploy a sample Java app with this tutorial.

The tutorial assumes that you have:

* A [verified Heroku Account](https://devcenter.heroku.com/articles/account-verification)
* [OpenJDK 17](https://www.azul.com/downloads/?version=java-17-lts&package=jdk#zulu) (or newer) installed locally
* [Postgres](https://devcenter.heroku.com/articles/heroku-postgresql#local-setup) installed locally
* An [Eco dynos plan](https://devcenter.heroku.com/articles/eco-dyno-hours) subscription (recommended)

If you prefer to u

# Introduction to Heroku for Java Developers

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/intro-for-java-developers)

Last updated December 16, 2019

## Table of Contents

* [Built-for-deployment applications](https://devcenter.heroku.com/articles/intro-for-java-developers#built-for-deployment-applications)
* [Enterprise Java applications on Heroku](https://devcenter.heroku.com/articles/intro-for-java-developers#enterprise-java-applications-on-heroku)
* [Summary](https://devcenter.heroku.com/articles/intro-for-java-developers#summary)

If you are an enterprise Java developer or server admin who develops, deploys and operates applications on traditional Java EE application servers, this guide will help you understand how to develop and deploy Java applications on Heroku.

## [Built-for-deployment applications](https://devcenter.heroku.com/articles/intro-for-java-developers#built-for-deployment-applications)

Heroku is a platform explicitly designed for development and deployment of applications. This is different from the “classic” software delivery process which flows something like this

* Development
* Packaging
* Distribution
* Install
* Deployment

Heroku is used by organizations who develop, deploy and operate an application within a single team or a few teams. There is no packaging, distribution or install element because the code never leaves the team/organization. This is a key reason why Heroku is more agile and once you have this in mind, it’s easier to decide how to best use Heroku as your deployment platform.

Here are some areas where a built-for-deployment application differs from a built-for-distribution application:

### [Version control is the central distribution mechanism](https://devcenter.heroku.com/articles/intro-for-java-developers#version-control-is-the-central-distribution-mechanism)

There is no reason to build a package of your code that can be consumed by external parties. No need for WAR, EAR, .exe installers or tarballs. You can freely choose how your application artifacts are structured. Typically, your runtime will execute your application directly from the file structure produced by your build process.

### [No need to externalize configuration from your code](https://devcenter.heroku.com/articles/intro-for-java-developers#no-need-to-externalize-configuration-from-your-code)

When you build an application meant for distribution, you often need to give your consumers a high degree of flexibility in how they configure your application during deployment. This has led to very sophisticated (and complex) frameworks for configuration often involving a large number of XML files. When the behavior of your application can be controlled to a large degree from external configuration, it becomes very hard to test and the expected behavior is harder to understand for the developer (and operator).

Built-for-deployment applications do not have this requirement and therefore you can eliminate all this complexity and instead perform most if not all configuration in your code. By doing configuration in code, you can take advantage of compile-time checks and testing generally becomes much easier. This leads to improved agility and robustness of your delivery process. You can see the trend towards this approach in how newer frameworks like [Guice](http://code.google.com/p/google-guice/) do not use configuration files and older frameworks like [Spring](http://www.springsource.org/) and [Java EE](https://www.oracle.com/technetwork/java/javaee/overview/index.html) are moving away from it.

### [Deployment is a highly automated pipeline process](https://devcenter.heroku.com/articles/intro-for-java-developers#deployment-is-a-highly-automated-pipeline-process)

Because the end-to-end lifecycle of the application is owned and controlled by a single team or organization, you can leverage version control and standard build automation tools to partially or completely automate the delivery process. This is another key factor in increasing agility and robustness of application delivery. The pipeline goes from code commit through a series of testing stages to release of the new version. The automation is often driven by the build system for the project assisted by tools like continuous integration servers.

## [Enterprise Java applications on Heroku](https://devcenter.heroku.com/articles/intro-for-java-developers#enterprise-java-applications-on-heroku)

The Java EE spec defines an application delivery process that is much closer to built-for-distribution than built-for-deployment. It introduces several concepts such as container-independent configuration, packaging, application installation and configuration, that are not necessary when you build applications purely for deployment. Over time, patterns and best practices have emerged for designing built-for-deployment applications that uses all the same Java EE APIs but without using the packaging, distribution, and configuration parts. These practices include

* The application is not deployed to an application server. Instead it bootstraps itself and configures all necessary services in the application code. It may or may not use an IoC container to assist with this configuration. [The Jetty web server](http://wiki.eclipse.org/Jetty/Tutorial/Embedding_Jetty) has been at the forefront of this approach. But [Tomcat](http://tomcat.apache.org) is also embeddable
* Libraries like the Servlet stack, JDBC drivers, JSP and other Java EE libraries are linked into the application using dependency management as opposed to being provided by the container.
* External configuration files are minimally used because there is really no need.
* Many Java EE services are still used even if there is no container, Servlet, JSP, JDBC and JMS are the most common ones.

Heroku follows this model for Java development and deployment. You can use as many or as few Java EE libraries as you need. But your application is organized for continuous deployment instead of packaging and distribution.

### [How do applications use Java EE APIs without a container?](https://devcenter.heroku.com/articles/intro-for-java-developers#how-do-applications-use-java-ee-apis-without-a-container)

Here are some examples of how you can use Java EE APIs:

* You can write Servlets and JSPs by using embedded Tomcat or Jetty library.
* You can use JSF and other rendering frameworks using Mojarra or MyFaces
* You can use javax.mail to build mail functionality on top of external SMTP services like those found in the [Heroku Add-ons catalog](https://elements.heroku.com/addons/)
* You can use JDBC to connect to databases like Heroku Postgres service or any one of the [MySQL add-ons](https://devcenter.heroku.com/articles?q=mysql)
* You can use Hibernate or DataNucleus JPA to provide an ORM persistence service on top of SQL databases or Database.com

### [How about the many other things that an application server provides?](https://devcenter.heroku.com/articles/intro-for-java-developers#how-about-the-many-other-things-that-an-application-server-provides)

The main purpose of application servers is to provide services for deployment and operations of applications. The list of features includes:

* Deployment
* Start/stop/restart
* Deployment of changes
* Clustering (scaling)
* Load-balancing
* Failover and high availability
* Logging
* Service binding

When you deploy an application to a cloud platform like Heroku, the cloud platform is responsible for delivering all these features. You don’t need to “bring your own WebLogic or WebSphere server” in order to get these features. They are a built-in part of the cloud platform.

Let’s take a look at each of these features and how Heroku works compared to a typical Java EE application server.

### [Deployment](https://devcenter.heroku.com/articles/intro-for-java-developers#deployment)

#### [Java EE application server](https://devcenter.heroku.com/articles/intro-for-java-developers#deployment-java-ee-application-server)

* Package as WAR or EAR
* Upload to server using a variety of tools
* Optionally perform additional configuration in server console

#### [Heroku](https://devcenter.heroku.com/articles/intro-for-java-developers#deployment-heroku)

[Git based workflow](http://www.heroku.com/how/deploy):

git push heroku master

### [Start/Stop/Restart](https://devcenter.heroku.com/articles/intro-for-java-developers#start-stop-restart)

#### [Java EE application server](https://devcenter.heroku.com/articles/intro-for-java-developers#start-stop-restart-java-ee-application-server)

Use the console or an API command to start/stop/restart an application. Application restarts are often error-prone because the server and the application runs in the same JVM.

#### [Heroku](https://devcenter.heroku.com/articles/intro-for-java-developers#start-stop-restart-heroku)

heroku restart

Applications run in separate JVMs, each running in its own [dyno](https://devcenter.heroku.com/articles/dynos), [completely isolated](https://devcenter.heroku.com/articles/dynos#isolation-and-security) from each other. This makes restarts very robust with zero side-effects on other applications as opposed to restarting a Java EE application running in the same process as other Java EE applications, sharing memory and other resources.

### [Deployment of changes](https://devcenter.heroku.com/articles/intro-for-java-developers#deployment-of-changes)

#### [Java EE application server](https://devcenter.heroku.com/articles/intro-for-java-developers#deployment-of-changes-java-ee-application-server)

* Package changes in WAR or EAR
* Upload to server using a variety of tools
* Use console or scripts to perform redeployment following one of several strategies to minimize downtime

#### [Heroku](https://devcenter.heroku.com/articles/intro-for-java-developers#deployment-of-changes-heroku)

[Redeployment](http://www.heroku.com/how/deploy) is simply another push of the code to Heroku:

git push heroku master

Heroku automatically builds the app, deploys it to new dynos and retires existing dynos.

### [Clustering (scaling)](https://devcenter.heroku.com/articles/intro-for-java-developers#clustering-scaling)

#### [Java EE application server](https://devcenter.heroku.com/articles/intro-for-java-developers#clustering-scaling-java-ee-application-server)

Complex and very app server specific configuration is needed. Clustering often relies on shared file systems like NAS or SAN. Java EE app servers often support stateful sessions with replication across the cluster, which requires additional configuration.

#### [Heroku](https://devcenter.heroku.com/articles/intro-for-java-developers#clustering-scaling-heroku)

Heroku applications are [scaled](https://devcenter.heroku.com/articles/scaling) with a single command:

heroku ps:scale web=2 worker=4 ...

Heroku takes care of spinning up and down nodes (dynos). Heroku does not support session replication. Heroku has a share-nothing architecture which makes it simple to configure and scale applications, but applications need to be architected for it. The application logic cannot rely on stateful sessions.

### [Load-balancing](https://devcenter.heroku.com/articles/intro-for-java-developers#load-balancing)

#### [Java EE application server](https://devcenter.heroku.com/articles/intro-for-java-developers#load-balancing-java-ee-application-server)

Load-balancing is most often accomplished using an external load balancer that must be separately configured. It’s usually the responsibility of an operations team to configure load balancers in front of clusters of app servers.

#### [Heroku](https://devcenter.heroku.com/articles/intro-for-java-developers#load-balancing-heroku)

Load-balancing is automatic. When you scale your application up and down, the individual nodes (dynos) automatically register and de-register with the [routing infrastructure](https://devcenter.heroku.com/articles/http-routing). Requests are routed to your application nodes (dynos) using round-robin. Heroku supports [session affinity](https://devcenter.heroku.com/articles/session-affinity), but it is not enabled by default.

### [Failover and high availability](https://devcenter.heroku.com/articles/intro-for-java-developers#failover-and-high-availability)

#### [Java EE application server](https://devcenter.heroku.com/articles/intro-for-java-developers#failover-and-high-availability-java-ee-application-server)

Application servers are often able to detect nodes that no longer respond and route traffic to other nodes. But they don’t automatically restart nodes. They also cannot respond to problems happening at the OS or hardware level. This requires additional monitoring tools. It is your own responsibility to integrate with your load-balancer so it can detect failed nodes.

#### [Heroku](https://devcenter.heroku.com/articles/intro-for-java-developers#failover-and-high-availability-heroku)

Heroku [detects if a node in an application crashes](https://devcenter.heroku.com/articles/dynos#dynos-and-requests) and then tries to restart it. A restart triggers provisioning of a completely new node which means that problems at the OS level or even hardware level can be overcome with a restart. Heroku separately monitors the hardware nodes and automatically retires broken instances, migrating nodes hosted by those instances to a fresh, new instance.

### [Logging](https://devcenter.heroku.com/articles/intro-for-java-developers#logging)

#### [Java EE application server](https://devcenter.heroku.com/articles/intro-for-java-developers#logging-java-ee-application-server)

Logging services are configured on the application server. The server can perform some routing of logs as well as log file rotation. Application servers sometimes also have UIs for searching and filtering logs. Applications must be properly configured to bind to and use the logging services configured on the server.

#### [Heroku](https://devcenter.heroku.com/articles/intro-for-java-developers#logging-heroku)

Applications are expected to send log events to standard out. Heroku streams log events received on standard out to the [Logplex](https://devcenter.heroku.com/articles/logging). Application owners can configure any number of “sinks” that consume these streams. There is no configuration required at the application level.

### [Service binding](https://devcenter.heroku.com/articles/intro-for-java-developers#service-binding)

#### [Java EE application server](https://devcenter.heroku.com/articles/intro-for-java-developers#service-binding-java-ee-application-server)

Application servers use a combination of deployment descriptors (multiple), server console and JNDI directory services to configure how applications bind to services.

#### [Heroku](https://devcenter.heroku.com/articles/intro-for-java-developers#service-binding-heroku)

[Service configuration](https://devcenter.heroku.com/articles/config-vars) is provided as OS environment variables. Applications are responsible for reading these variables and properly configure connectors based on the connection parameters. Configuration files such as spring config files can be used for this, but a simpler approach is to perform the configuration from a bootstrap code segment.

## [Summary](https://devcenter.heroku.com/articles/intro-for-java-developers#summary)

If you’ve made it this far, you should have a pretty good idea of how Heroku works compared to classic enterprise Java development. The best way to learn more is to try it out:

* [Getting started with Java on Heroku](https://devcenter.heroku.com/articles/getting-started-with-java)
* [Deploying Spring Boot Applications to Heroku](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku)

### [Keep reading](https://devcenter.heroku.com/articles/intro-for-java-developers#keep-reading)

* [Java](https://devcenter.heroku.com/categories/java-support)

### [Feedback](https://devcenter.heroku.com/articles/intro-for-java-developers#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fintro-for-java-developers&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Deploying Java Apps on Heroku

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/deploying-java)

Last updated September 29, 2023

## Table of Contents

* [Prerequisites](https://devcenter.heroku.com/articles/deploying-java#prerequisites)
* [Overview](https://devcenter.heroku.com/articles/deploying-java#overview)
* [Verify that your pom.xml file is set up correctly](https://devcenter.heroku.com/articles/deploying-java#verify-that-your-pom-xml-file-is-set-up-correctly)
* [Specify a JDK](https://devcenter.heroku.com/articles/deploying-java#specify-a-jdk)
* [The Procfile](https://devcenter.heroku.com/articles/deploying-java#the-procfile)
* [How to keep build artifacts out of git](https://devcenter.heroku.com/articles/deploying-java#how-to-keep-build-artifacts-out-of-git)
* [Build your app and run it locally](https://devcenter.heroku.com/articles/deploying-java#build-your-app-and-run-it-locally)
* [Deploy your application to Heroku](https://devcenter.heroku.com/articles/deploying-java#deploy-your-application-to-heroku)

This article describes how to take an existing Java app and deploy it to Heroku.

If you are new to Heroku, you might want to start with the [Getting Started with Java on Heroku](https://devcenter.heroku.com/articles/getting-started-with-java) tutorial.

## [Prerequisites](https://devcenter.heroku.com/articles/deploying-java#prerequisites)

The best practices in this article assume that you have:

* an existing Java app that uses Maven as a build tool.
* a free [Heroku account](https://signup.heroku.com/signup/dc)
* the [Heroku CLI](https://cli.heroku.com/)
* a Java JDK
* [Maven 3](http://maven.apache.org/download.cgi)

## [Overview](https://devcenter.heroku.com/articles/deploying-java#overview)

The details of Heroku’s Java Support are described in the [Heroku Java Support](https://devcenter.heroku.com/articles/java-support#activation) article.

Heroku Java support for Maven will be applied to applications that contain a pom.xml file.

## [Verify that your pom.xml file is set up correctly](https://devcenter.heroku.com/articles/deploying-java#verify-that-your-pom-xml-file-is-set-up-correctly)

If your app has any dependencies, the pom.xml file should include the maven-dependency-plugin. It tells Maven to copy the jar files that your app depends on to the target/dependency directory. This way, they are put into the slug, and the .m2 directory can be removed from the slug. It should look something like this:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<version>1.0-SNAPSHOT</version>

<artifactId>helloworld</artifactId>

<dependencies>

...

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-dependency-plugin</artifactId>

<version>3.0.1</version>

<executions>

<execution>

<id>copy-dependencies</id>

<phase>package</phase>

<goals><goal>copy-dependencies</goal></goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

## [Specify a JDK](https://devcenter.heroku.com/articles/deploying-java#specify-a-jdk)

Optionally, you can specify a JDK. For more information, see [Specifying a Java version](https://devcenter.heroku.com/articles/java-support#specifying-a-java-version).

## [The Procfile](https://devcenter.heroku.com/articles/deploying-java#the-procfile)

A [Procfile](https://devcenter.heroku.com/articles/procfile) is a text file in the root directory of your application, that defines process types and explicitly declares what command should be executed to start your app. Your Procfile will look something like this:

web: java $JAVA\_OPTS -cp target/classes:target/dependency/\* com.example.HelloWorld

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.

The command in a web process type must bind to the port number [specified in the PORT environment variable](https://devcenter.heroku.com/articles/dynos#local-environment-variables). If it does not, the dyno will not start.

## [How to keep build artifacts out of git](https://devcenter.heroku.com/articles/deploying-java#how-to-keep-build-artifacts-out-of-git)

Prevent build artifacts from going into revision control by creating a .gitignore file. Here’s a typical .gitignore file:

target

## [Build your app and run it locally](https://devcenter.heroku.com/articles/deploying-java#build-your-app-and-run-it-locally)

To build your app locally do this:

Use the Git Bash application to open a command shell on Windows. A shortcut for this application was added to your desktop as part of the CLI installation.

mvn clean install

heroku local --port 5001

Your app should now be running on <http://localhost:5001/>.

## [Deploy your application to Heroku](https://devcenter.heroku.com/articles/deploying-java#deploy-your-application-to-heroku)

After you commit your changes to git, you can deploy your app to Heroku.

git add .

git commit -m "Added a Procfile."

heroku login

Enter your Heroku credentials.

...

heroku create

Creating arcane-lowlands-8408... done, stack is heroku-18

http://arcane-lowlands-8408.herokuapp.com/ | git@heroku.com:arcane-lowlands-8408.git

Git remote heroku added

git push heroku main

...

-----> Java app detected

...

-----> Launching... done

http://arcane-lowlands-8408.herokuapp.com deployed to Heroku

To open the app in your browser, type heroku open.

### [Keep reading](https://devcenter.heroku.com/articles/deploying-java#keep-reading)

* [Java](https://devcenter.heroku.com/categories/java-support)

### [Feedback](https://devcenter.heroku.com/articles/deploying-java#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fdeploying-java&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Preparing a Java Web App for Production on Heroku

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/preparing-a-java-web-app-for-production-on-heroku)

Last updated June 14, 2023

## Table of Contents

* [Force the use of HTTPS](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#force-the-use-of-https)
* [Rate-limit API calls](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#rate-limit-api-calls)
* [Use a distributed session store](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#use-a-distributed-session-store)
* [Enable JVM Runtime Metrics](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#enable-jvm-runtime-metrics)
* [Configure alerting](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#configure-alerting)
* [Configure error and maintenance pages](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#configure-error-and-maintenance-pages)
* [Attach a logging add-on](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#attach-a-logging-add-on)
* [Attach an error-tracking add-on](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#attach-an-error-tracking-add-on)
* [Attach a vulnerability detection add-on](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#attach-a-vulnerability-detection-add-on)

It is important to ensure that an app is secure, scalable, and resilient to failure before sending it to production. This guide provides an overview of important steps required to make a Java web application production-ready on Heroku. If you are using Spring Boot, you might prefer the article on [Preparing a Spring Boot App for Production](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku).

## [Force the use of HTTPS](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#force-the-use-of-https)

Unless you have very specific needs, your app should be using HTTPS for all requests. Heroku provides an HTTPS URL (of the form https://<app-name>-<random-identifier>.herokuapp.com) for every app, as well as free tools for [adding your own domains](https://devcenter.heroku.com/articles/custom-domains) and [certificates](https://devcenter.heroku.com/articles/understanding-ssl-on-heroku).

You can enforce the use of HTTPS when your app is running on Heroku by creating a Servlet filter in your app with the following code:

public class HttpsEnforcer implements Filter {

public static final String X\_FORWARDED\_PROTO = "X-Forwarded-Proto";

@Override

public void init(FilterConfig filterConfig) throws ServletException {}

@Override

public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {

HttpServletRequest request = (HttpServletRequest) servletRequest;

HttpServletResponse response = (HttpServletResponse) servletResponse;

if (request.getHeader(X\_FORWARDED\_PROTO) != null) {

if (request.getHeader(X\_FORWARDED\_PROTO).indexOf("https") != 0) {

String pathInfo = (request.getPathInfo() != null) ? request.getPathInfo() : "";

response.sendRedirect("https://" + request.getServerName() + pathInfo);

return;

}

}

filterChain.doFilter(request, response);

}

@Override

public void destroy() { }

}

This configuration instructs the servlet container to redirect all plain HTTP requests back to the same URL using HTTPS if the X-Forwarded-Proto header is present. [Heroku sets the X-Forwarded-Proto header](https://devcenter.heroku.com/articles/http-routing#heroku-headers) for you, which means the request will be redirected back through the Heroku router where SSL is terminated. In your localhost environment, you can continue to use plain HTTP.

## [Rate-limit API calls](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#rate-limit-api-calls)

Rate limiting is the process of controlling traffic to a server based on client IPs, blocked IPs, geolocation, and other factors. One of the most popular rate-limiting libraries for Java is [Bucket4j](https://github.com/vladimir-bukhtoyarov/bucket4j), which can be used by creating another Servlet filter in your application. First, add the following dependency to your pom.xml:

<dependency>

<groupId>com.github.vladimir-bukhtoyarov</groupId>

<artifactId>bucket4j-core</artifactId>

</dependency>

The create a Servlet filter with the following code:

public class ThrottlingFilter implements javax.servlet.Filter {

private Bucket createNewBucket() {

long overdraft = 50;

Refill refill = Refill.greedy(10, Duration.ofSeconds(1));

Bandwidth limit = Bandwidth.classic(overdraft, refill);

return Bucket4j.builder().addLimit(limit).build();

}

@Override

public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {

HttpServletRequest httpRequest = (HttpServletRequest) servletRequest;

HttpSession session = httpRequest.getSession(true);

String appKey = SecurityUtils.getThirdPartyAppKey();

Bucket bucket = (Bucket) session.getAttribute("throttler-" + appKey);

if (bucket == null) {

Bucket bucket = createNewBucket();

session.setAttribute("throttler-" + appKey, bucket);

}

if (bucket.tryConsume(1)) {

filterChain.doFilter(servletRequest, servletResponse);

} else {

HttpServletResponse httpResponse = (HttpServletResponse) servletResponse;

httpResponse.setContentType("text/plain");

httpResponse.setStatus(429);

httpResponse.getWriter().append("Too many requests");

}

}

}

Other examples can be found in the [Bucket4j documentation](https://github.com/vladimir-bukhtoyarov/bucket4j/blob/master/asciidoc/src/main/docs/asciidoc/about.adoc).

## [Use a distributed session store](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#use-a-distributed-session-store)

Storing sessions in-memory inhibits the [disposability](https://12factor.net/disposability) and horizontal scalability of dynos. That’s why it’s important to instead use a distributed session store like Redis.

There are many ways to implement session storage with Java and Redis that depend on your server and frameworks. Heroku recommends using [Redisson](https://github.com/redisson/redisson), which works with most popular options, as described in this article on [Java Session Handling](https://devcenter.heroku.com/articles/java-session-handling-on-heroku).

## [Enable JVM Runtime Metrics](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#enable-jvm-runtime-metrics)

The [JVM Runtime Metrics](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm) feature allows you to view heap memory, non-heap memory, and garbage collection activity for any app that runs inside of the Java Virtual Machine (JVM). The feature is safe to use in production and can help you identify performance-related problems.

## [Configure alerting](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#configure-alerting)

When your application experiences a problem, it should alert a human. At a minimum, the app should:

* Alert a human if it is down
* Alert a human if error rates exceed specific thresholds
* Alert a human if latency is high

Heroku’s [Threshold Alerting](https://devcenter.heroku.com/articles/metrics#threshold-alerting) feature is available to apps running on Professional dynos. You can also choose from the many [alerting and monitoring add-ons](https://elements.heroku.com/addons) in the Heroku Add-on Marketplace.

## [Configure error and maintenance pages](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#configure-error-and-maintenance-pages)

Heroku lets you configure the static HTML pages that are shown to users when your application experiences an error (such as crashing) or goes down for maintenance. Please see [Customizing Error Pages](https://devcenter.heroku.com/articles/error-pages#customize-pages) for more information.

## [Attach a logging add-on](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#attach-a-logging-add-on)

By default, Heroku stores 1500 lines of logs from your application. However, it makes the full log stream available as a service that several add-on providers consume to provide features such as log persistence, search, and alerts via email or SMS.

You can provision one of these logging add-ons, Papertrail, by running the following command on your app:

heroku addons:create papertrail

To see Papertrail in action, visit your application’s Heroku URL a few times. Each visit generates more log messages, which should be routed to the add-on.

## [Attach an error-tracking add-on](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#attach-an-error-tracking-add-on)

While your application logs capture the regular activity of your server processes, an error-tracking add-on can capture more detail for exceptional cases. This can be useful in identifying common problems your users encounter, or to diagnose poor performance. The Add-on Marketplace has a number of options including the Rollbar service, which you can add to your app by running:

heroku addons:create rollbar

After following the guide for [setting up Rollbar integration](https://devcenter.heroku.com/articles/rollbar), you’ll begin to see a record of all errors with their associated stack traces and other details coming from your app.

## [Attach a vulnerability detection add-on](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#attach-a-vulnerability-detection-add-on)

The final add-on every production app should use is one that detects security vulnerabilities in your source code. One such service is [Snyk](https://devcenter.heroku.com/articles/snyk), which you can attach to your app by running:

heroku addons:create snyk

Snyk scans your source code and compares the dependencies in either your pom.xml or build.gradle file against its database of known security vulnerabilities. If it finds any, it sends a notification that describes how to resolve them.

### [Keep reading](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#keep-reading)

* [Java](https://devcenter.heroku.com/categories/java-support)

### [Feedback](https://devcenter.heroku.com/articles/preparing-a-java-web-app-for-production-on-heroku#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fpreparing-a-java-web-app-for-production-on-heroku&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Frequently Asked Questions About Java

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/java-faq)

Last updated November 07, 2022

## Table of Contents

* [What kind of skill sets are required to build Java applications on Heroku (JEE, Spring, etc)?](https://devcenter.heroku.com/articles/java-faq#what-kind-of-skill-sets-are-required-to-build-java-applications-on-heroku-jee-spring-etc)
* [Can I deploy standard Java web applications to Heroku?](https://devcenter.heroku.com/articles/java-faq#can-i-deploy-standard-java-web-applications-to-heroku)
* [Do Java applications run in a JEE container on Heroku?](https://devcenter.heroku.com/articles/java-faq#do-java-applications-run-in-a-jee-container-on-heroku)
* [Can I deploy an application packaged as a WAR file to Heroku?](https://devcenter.heroku.com/articles/java-faq#can-i-deploy-an-application-packaged-as-a-war-file-to-heroku)
* [Can I run a stand-alone Java application (that is not a web application) on Heroku?](https://devcenter.heroku.com/articles/java-faq#can-i-run-a-stand-alone-java-application-that-is-not-a-web-application-on-heroku)
* [Can I spawn and control threads?](https://devcenter.heroku.com/articles/java-faq#can-i-spawn-and-control-threads)
* [Can I read from and write to the file system?](https://devcenter.heroku.com/articles/java-faq#can-i-read-from-and-write-to-the-file-system)
* [What happens to data written to standard out?](https://devcenter.heroku.com/articles/java-faq#what-happens-to-data-written-to-standard-out)
* [Is there any constraints on using the core Java APIs?](https://devcenter.heroku.com/articles/java-faq#is-there-any-constraints-on-using-the-core-java-apis)
* [How do I specify which JDK I would like my application to use?](https://devcenter.heroku.com/articles/java-faq#how-do-i-specify-which-jdk-i-would-like-my-application-to-use)
* [Do I need to push my source code to Heroku?](https://devcenter.heroku.com/articles/java-faq#do-i-need-to-push-my-source-code-to-heroku)
* [Are there benefits to pushing my source code to Heroku?](https://devcenter.heroku.com/articles/java-faq#are-there-benefits-to-pushing-my-source-code-to-heroku)
* [Can I use other build systems than Maven?](https://devcenter.heroku.com/articles/java-faq#can-i-use-other-build-systems-than-maven)
* [How do I build Force.com and Database.com Java applications on Heroku?](https://devcenter.heroku.com/articles/java-faq#how-do-i-build-force-com-and-database-com-java-applications-on-heroku)
* [What constraints should I be aware of when developing applications on Heroku?](https://devcenter.heroku.com/articles/java-faq#what-constraints-should-i-be-aware-of-when-developing-applications-on-heroku)
* [Can I Use Heroku’s Built in PostgreSQL Databases With Java?](https://devcenter.heroku.com/articles/java-faq#can-i-use-heroku-s-built-in-postgresql-databases-with-java)

## [What kind of skill sets are required to build Java applications on Heroku (JEE, Spring, etc)?](https://devcenter.heroku.com/articles/java-faq#what-kind-of-skill-sets-are-required-to-build-java-applications-on-heroku-jee-spring-etc)

You can deploy any Java application on Heroku. It is not limited to JEE or other frameworks. You can deploy Java web applications packaged as WAR files using [WAR deployment](https://devcenter.heroku.com/articles/war-deployment) or you can [deploy Maven based Java projects](https://devcenter.heroku.com/articles/getting-started-with-java) of any kind using the git versioning system. In both cases you are free to use any frameworks and libraries that you choose, including Spring and JEE components such as Servlets, JSPs, JDBC drivers, etc.

## [Can I deploy standard Java web applications to Heroku?](https://devcenter.heroku.com/articles/java-faq#can-i-deploy-standard-java-web-applications-to-heroku)

Yes. You can build and deploy Java web applications that use all the common APIs: Servlet, JSP, JDBC, taglibs, JSF etc. You can deploy Java web applications in two ways. You can build it locally and [deploy as a WAR package](https://devcenter.heroku.com/articles/war-deployment) to Heroku or you can set it up as a [Maven WAR project that includes an embedded web app runner](https://devcenter.heroku.com/articles/java-webapp-runner) and deploy to Heroku using git. The former is a more familiar approach to most Java developers. The latter is better optimized for continuous delivery.

## [Do Java applications run in a JEE container on Heroku?](https://devcenter.heroku.com/articles/java-faq#do-java-applications-run-in-a-jee-container-on-heroku)

Applications deployed as WAR files run in a Tomcat container. Applications deployed using Git are not deployed to a container. Instead you bundle in a web server library like Jetty or embedded Tomcat and the application can execute as a self-contained unit. Frameworks like [Thorntail](https://thorntail.io) do this for you. The latter approach gives you the most optimal setup for continuous delivery with the best control over changes to the environment. The former is a more common approach that most Java developers are familiar with.

There are some [3rd party buildpacks for installing JEE containers](https://github.com/jkutner/heroku-buildpack-wildfly) on Heroku.

## [Can I deploy an application packaged as a WAR file to Heroku?](https://devcenter.heroku.com/articles/java-faq#can-i-deploy-an-application-packaged-as-a-war-file-to-heroku)

Yes. See our article on [deploying WAR files](https://devcenter.heroku.com/articles/war-deployment).

## [Can I run a stand-alone Java application (that is not a web application) on Heroku?](https://devcenter.heroku.com/articles/java-faq#can-i-run-a-stand-alone-java-application-that-is-not-a-web-application-on-heroku)

Yes. In fact, to Heroku there is no difference between a stand-alone application and a web application. They are both Java processes. The web application happens to listen for web requests on a TCP port and process them using some framework like the Servlet API.

Your application can be run either as a worker process (that never exits), a one-off batch script that you launch with the heroku run command, or as a scheduled process that gets executed by the [Heroku Scheduler](https://elements.heroku.com/addons/scheduler).

## [Can I spawn and control threads?](https://devcenter.heroku.com/articles/java-faq#can-i-spawn-and-control-threads)

Yes.

## [Can I read from and write to the file system?](https://devcenter.heroku.com/articles/java-faq#can-i-read-from-and-write-to-the-file-system)

Yes. But the file system is [ephemeral](https://devcenter.heroku.com/articles/dynos#isolation-and-security). Any data you write to the file system will not be available to other dynos of your application. Generally you can assume that a file written in the beginning of a request will be available for the duration of the request, subject to the previous constraint (your dyno may get restarted in the middle of a request). Do not rely on the local filesystem for any data you want to keep around.

## [What happens to data written to standard out?](https://devcenter.heroku.com/articles/java-faq#what-happens-to-data-written-to-standard-out)

Standard out is piped to the [Heroku logging service](https://devcenter.heroku.com/articles/logging). You can use the heroku logs command to retrieve this log stream (in streaming mode if you add the –tail options). You can also add syslog sinks that will retrieve the log stream and can store it in log files or index it for searching and reporting.

## [Is there any constraints on using the core Java APIs?](https://devcenter.heroku.com/articles/java-faq#is-there-any-constraints-on-using-the-core-java-apis)

No. Your application will be executed using a recent version of [OpenJDK](http://openjdk.java.net/) with no modifications.

## [How do I specify which JDK I would like my application to use?](https://devcenter.heroku.com/articles/java-faq#how-do-i-specify-which-jdk-i-would-like-my-application-to-use)

Heroku provides a declarative way to specify your Java version from within your application. See [Specifying a Java version](https://devcenter.heroku.com/articles/java-support#specifying-a-java-version) for more information.

## [Do I need to push my source code to Heroku?](https://devcenter.heroku.com/articles/java-faq#do-i-need-to-push-my-source-code-to-heroku)

No, Heroku will build your app for you if you push up your source code, but if you’d like to build your application to a WAR file on your own you can also [deploy WAR files](https://devcenter.heroku.com/articles/war-deployment) directly to Heroku.

## [Are there benefits to pushing my source code to Heroku?](https://devcenter.heroku.com/articles/java-faq#are-there-benefits-to-pushing-my-source-code-to-heroku)

Yes, there are numerous benefits to pushing the contents of your Git repository to the platform:

* Differences between the build environment and the runtime environment are a common source of bad deploys in traditional deployment environments. Doing the app’s build in the same environment that the app will later run greatly reduces this risk.
* Pushing code rather than builds gives you and your team greater visibility into what code is deployed where. For example, the command git diff production/master staging/master will show the exact code differences between staging and production.
* Git is highly optimized for transmitting only what has changed. This means that most code pushes (after the first one) will take only seconds, instead of many minutes that it may take to transfer a full build artifact.
* Deploying with revision control makes for smoother collaboration between team members with deploy rights. For example, it provides an overrideable safeguard against accidentally overwriting a more recent deploy with an older one.

## [Can I use other build systems than Maven?](https://devcenter.heroku.com/articles/java-faq#can-i-use-other-build-systems-than-maven)

If you’d like Heroku to build your application for you, then you can choose from Maven or Gradle. But Ant based builds are available through 3rd party contributed buildpacks like [this one](https://github.com/dennisg/heroku-buildpack-ant). When taking advantage of our Maven support you must specify a pom.xml file at the top level of your project. If you prefer, you can use Maven to kick off Ant scripts and other types of build scripts.

If you’re deploying a WAR file you can, of course, build it with any build system you’d like.

Any other type of build can be supported on Heroku via [custom Buildpacks](https://devcenter.heroku.com/articles/buildpacks)

## [How do I build Force.com and Database.com Java applications on Heroku?](https://devcenter.heroku.com/articles/java-faq#how-do-i-build-force-com-and-database-com-java-applications-on-heroku)

Use the [Database.com SDK for Java](https://forcedotcom.github.io/java-sdk/force-sdk-overview.html).

## [What constraints should I be aware of when developing applications on Heroku?](https://devcenter.heroku.com/articles/java-faq#what-constraints-should-i-be-aware-of-when-developing-applications-on-heroku)

Here are some things to keep in mind when designing applications on Heroku

* Your application source code plus built artifacts must be less than 500 MB when compressed. Use .slugignore to prevent files in your git repo from being included in the deployed package
* You can have many process types in a single application, but only one of those types can be a web process that receives requests from the routing layer.
* The web process must listen on one and only one port. The port must be the one specified in the $PORT variable. If your process listens on other ports, it will be shut down by Heroku.
* The Heroku routing infrastructure does not support “sticky sessions”. Requests from clients will be distributed randomly to all dynos running your application.
* Individual dyno processes of the same process type (e.g. the web process type) cannot communicate directly with each other. For example, they cannot replicate state between them. Heroku is a share-nothing architecture where each node is completely isolated.
* A single dyno (for example, a single instance of your web process type) may be restarted by Heroku at any point in time. Your application must be designed to anticipate restarts without losing data or affecting the user experience in a material way. Your dyno will receive a SIGTERM signal before it is killed. You can trap this signal to perform an orderly shutdown.
* Any data you write to the file system will not be available to other dynos of your application. Generally you can assume that a file written in the beginning of a request will be available for the duration of the request, subject to the previous constraint (your dyno may get restarted in the middle of a request). Do not rely on the local filesystem for any data you want to keep around.
* Your application must boot in one minute or less.
* You can increase the memory setting for your application from the default specified in the JAVA\_OPTS config variable. But performance will eventually suffer. You should always design for horizontal scale-out when possible instead of relying on increasing the heap. You can check the setting with heroku config

## [Can I Use Heroku’s Built in PostgreSQL Databases With Java?](https://devcenter.heroku.com/articles/java-faq#can-i-use-heroku-s-built-in-postgresql-databases-with-java)

Yes, You can connect to either a shared or Heroku Postgres database from Java. You can use JDBC or any other means of database connectivity that you’re used to. You can also connect to a Heroku Postgres database from your local machine to troubleshoot your application. The Dev Center has more information on [Heroku Postgres](https://devcenter.heroku.com/categories/heroku-postgres).

### [Keep reading](https://devcenter.heroku.com/articles/java-faq#keep-reading)

* [Java](https://devcenter.heroku.com/categories/java-support)

### [Feedback](https://devcenter.heroku.com/articles/java-faq#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fjava-faq&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Heroku Java Support

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/java-support)

Last updated October 19, 2023

## Table of Contents

* [Activation](https://devcenter.heroku.com/articles/java-support#activation)
* [Build Behavior](https://devcenter.heroku.com/articles/java-support#build-behavior)
* [Environment](https://devcenter.heroku.com/articles/java-support#environment)
* [Supported Java Versions](https://devcenter.heroku.com/articles/java-support#supported-java-versions)
* [Postgres Auto-Provisioning](https://devcenter.heroku.com/articles/java-support#postgres-auto-provisioning)
* [Further Reading](https://devcenter.heroku.com/articles/java-support#further-reading)

Heroku can run Java applications across a variety of Java implementations and includes support for framework-specific workflows.

This document describes the general behavior of Heroku as it relates to the recognition and execution of Java applications. General Java support on Heroku refers to the support for all frameworks except for Play. You can read about Play framework support in the [Play framework support reference](https://devcenter.heroku.com/articles/play-support).

For framework-specific tutorials visit:

* [Getting Started with Java on Heroku](https://devcenter.heroku.com/articles/getting-started-with-java)
* [Java Tutorials](https://devcenter.heroku.com/categories/java-support)

## [Activation](https://devcenter.heroku.com/articles/java-support#activation)

The default build system for Java applications on Heroku is Maven. Heroku Java support for Maven is applied to applications that contain a pom.xml.

When a deployed application is recognized as a Java application, Heroku responds with Java app detected.

git push heroku master

-----> Java app detected

## [Build Behavior](https://devcenter.heroku.com/articles/java-support#build-behavior)

The following command is run to build your app:

mvn -B -DskipTests clean dependency:list install

However, if Heroku detects an mvnw script in your application’s repository, it runs this script instead of the default Maven installation. You can override this behavior by [explicitly setting a Maven version](https://devcenter.heroku.com/articles/java-support#specifying-a-maven-version).

The maven repo is cached between builds to improve performance.

## [Environment](https://devcenter.heroku.com/articles/java-support#environment)

The following environment variables are set in dyno at boot-time:

* PORT: The web process binds to this HTTP port
* JAVA\_HOME: The location of the JDK install directory
* LD\_LIBRARY\_PATH: With the location of the JDK shared libraries
* JDBC\_DATABASE\_URL: If a DATABASE\_URL variable is present, this variable is populated with the converted form. See [Connecting to Relational Databases on Heroku with Java](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java) for more information.
* JAVA\_TOOL\_OPTIONS: Default Java options based on dyno size
* JAVA\_OPTS: Default Java options based on dyno size (identical to JAVA\_TOOL\_OPTIONS)

### [About JAVA\_TOOL\_OPTIONS](https://devcenter.heroku.com/articles/java-support#about-java_tool_options)

JAVA\_TOOL\_OPTIONS is directly supported by Java and intended to [augment a command line in environments where the command line can’t be accessed or modified](http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#tooloptions). Heroku uses this value to set default Java options based on dyno size. Since Java automatically picks it up, you don’t need to include it in your Procfile command.

You can override these settings in the Procfile command, which takes precedence over the defaults. For example, to change the default of -Xmx300m, you could pass in:

web: java -Xms150M -jar target/myapp.jar

You can also set your own JAVA\_TOOL\_OPTIONS config var. Setting your own causes its value to be appended to Heroku’s defaults and take precedence. Individual options not overridden in the Procfile command or custom JAVA\_TOOL\_OPTIONS are still in effect.

When a Java process is started on your dyno, the following Java options are added to JAVA\_TOOL\_OPTIONS and automatically picked up by Java:

* -Dfile.encoding=UTF-8
* -XX:+UseContainerSupport (for Java 11 and higher)

### [Adjusting Environment for a Dyno Size](https://devcenter.heroku.com/articles/java-support#adjusting-environment-for-a-dyno-size)

When a new [dyno type](https://devcenter.heroku.com/articles/dyno-types) is selected, the following settings are automatically added to JAVA\_TOOL\_OPTIONS:

* eco, basic, or standard-1x: -Xmx300m -Xss512k -XX:CICompilerCount=2
* standard-2x: -Xmx671m -XX:CICompilerCount=2
* performance-m: -Xmx2g
* performance-l: -Xmx12g

For Private Space dynos, the values are:

* private-s: -Xmx671m -XX:CICompilerCount=2
* private-m: -Xmx2g
* private-l: -Xmx12g

### [Monitoring Resource Usage](https://devcenter.heroku.com/articles/java-support#monitoring-resource-usage)

You can use additional JVM flags to monitor resource usage in a dyno. The following flags are recommended for monitoring resource usage:

-XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+UseConcMarkSweepGC

See the [troubleshooting](https://devcenter.heroku.com/articles/java-memory-issues) article for more information about tuning a JVM process.

## [Supported Java Versions](https://devcenter.heroku.com/articles/java-support#supported-java-versions)

Heroku currently uses **OpenJDK 8** to run your application by default. Other OpenJDK versions are also available. Depending on the OpenJDK version you select the latest available version of that JDK uses each time you deploy your app.

Current default versions for each OpenJDK version and Heroku stack are:

| **OpenJDK Version** | [**heroku-20**](https://devcenter.heroku.com/articles/heroku-20-stack) | [**heroku-22**](https://devcenter.heroku.com/articles/heroku-22-stack) |
| --- | --- | --- |
| OpenJDK 7 | 1.7.0\_352 | 1.7.0\_352 |
| OpenJDK 8 | 1.8.0\_392 | 1.8.0\_392 |
| OpenJDK 11 | 11.0.21 | 11.0.21 |
| OpenJDK 13 | 13.0.14 | 13.0.14 |
| OpenJDK 14 | 14.0.2 | - |
| OpenJDK 15 | 15.0.10 | 15.0.10 |
| OpenJDK 16 | 16.0.2 | - |
| OpenJDK 17 | 17.0.9 | 17.0.9 |
| OpenJDK 18 | 18.0.2.1 | 18.0.2.1 |
| OpenJDK 19 | 19.0.2 | 19.0.2 |
| OpenJDK 20 | 20.0.2 | 20.0.2 |
| OpenJDK 21 | 21.0.1 | 21.0.1 |

Rows marked up with red text and background indicate an OpenJDK version that is **fully end-of-life and no longer receiving updates** of any kind from the upstream maintainers and is no longer supported by Heroku.

The OpenJDK that your app uses is included in the slug, which affects your slug size.

### [Specifying a Java Version](https://devcenter.heroku.com/articles/java-support#specifying-a-java-version)

You can specify a Java version by adding a file called system.properties to your application.

Set a property java.runtime.version in the file:

java.runtime.version=11

Refer to the [supported Java versions table](https://devcenter.heroku.com/articles/java-support#supported-java-versions) for accepted major version values. Because the default is 8, you don’t need this file to use Java 8.

You can also pin your JDK update version by using a value such as this:

java.runtime.version=1.8.0\_202

However, we encourage you to use the more general 1.8 format, which automatically installs any security updates.

### [Specifying an OpenJDK Distribution](https://devcenter.heroku.com/articles/java-support#specifying-an-openjdk-distribution)

Heroku supports builds of OpenJDK from either [Azul® Zulu®](https://www.azul.com/downloads/?package=jdk#download-openjdk) or Heroku. Both distributions are built from the same source and are fully compliant with the Java SE specification. Azul® Zulu® builds of OpenJDK are also [TCK/JCK](https://en.wikipedia.org/wiki/Technology_Compatibility_Kit) certified and are our recommended OpenJDK distribution.

Current default distributions of OpenJDK for each stack are:

| [**heroku-18**](https://devcenter.heroku.com/articles/heroku-18-stack) | [**heroku-20**](https://devcenter.heroku.com/articles/heroku-20-stack) | [**heroku-22**](https://devcenter.heroku.com/articles/heroku-22-stack) |
| --- | --- | --- |
| Heroku | Heroku | [Azul® Zulu®](https://www.azul.com/downloads/?package=jdk#download-openjdk) |

#### [Using a Non-default Distribution](https://devcenter.heroku.com/articles/java-support#using-a-non-default-distribution)

To use a non-default distribution with your app, you must [specify a Java version](https://devcenter.heroku.com/articles/java-support#specifying-a-java-version). Prefix the version string with either heroku- or zulu- to select the respective distribution.

To use OpenJDK 11 and explicitly select Azul® Zulu® as the OpenJDK distribution, add the following to system.properties:

java.runtime.version=zulu-11

### [Upgrading your Java Version](https://devcenter.heroku.com/articles/java-support#upgrading-your-java-version)

All Java apps are automatically upgraded to the latest available JDK version when and only when they’re deployed. They aren’t upgraded if the app isn’t redeployed or if a specific version is configured in the system.properties file.

### [Specifying a Maven Version](https://devcenter.heroku.com/articles/java-support#specifying-a-maven-version)

Heroku supports the [Maven Wrapper](https://maven.apache.org/wrapper/), which is the recommended mechanism for defining a Maven version. If Heroku detects an mvnw file in the root directory of your repository, it uses this script to launch the Maven process.

You can also specify a Maven version with the system.properties file by setting a maven.version property like this:

maven.version=3.9.4

If this property is defined, the mvnw script is ignored. You are not upgraded to a newer version automatically and need to update your system.properties file to get a newer version.

The default, if you don’t specify a version and don’t use Maven Wrapper, is 3.9.4. This default will change over time and we recommend using [Maven Wrapper](https://maven.apache.org/wrapper/) to ensure stable builds.

### [Default Web Process Type](https://devcenter.heroku.com/articles/java-support#default-web-process-type)

The Java buildpack automatically detects the use of the [Spring Boot](https://spring.io/projects/spring-boot) and [Thorntail](https://thorntail.io/) web frameworks. For Spring Boot, it creates a web process type with the following command:

java -Dserver.port=$PORT $JAVA\_OPTS -jar target/\*.jar

For Thorntail, the buildpack uses this command for the default web process type:

java -Dswarm.http.port=$PORT $JAVA\_OPTS -jar target/\*.jar

You can override these defaults or define a custom process type using a [Procfile](https://devcenter.heroku.com/articles/procfile). The appropriate command depends on your app and the frameworks in use. See one of the [Java tutorials](https://devcenter.heroku.com/categories/java-support) for information on setting up your Procfile.

## [Postgres Auto-Provisioning](https://devcenter.heroku.com/articles/java-support#postgres-auto-provisioning)

This section is only applicable to accounts created before [May 15, 2023](https://devcenter.heroku.com/changelog-items/2596) or if you [asked Heroku Support to enable auto-provisioning for your account](https://help.heroku.com/N4M2HO67/how-do-i-toggle-auto-provisioning-of-a-database-on-first-deployment).

A mini Heroku Postgres database automatically provisions on the first deploy of your Java applications. These apps must have a dependency on the [Postgres JDBC driver](https://jdbc.postgresql.org) or [pgjdbc-ng driver](https://impossibl.github.io/pgjdbc-ng/) in their pom.xml. This auto-provisioning populates the DATABASE\_URL environment variable.

If you don’t want the Postgres add-on, remove it by running:

$ heroku addons:destroy DATABASE --app example-app

## [Further Reading](https://devcenter.heroku.com/articles/java-support#further-reading)

* [Warming Up a Java Process](https://devcenter.heroku.com/articles/warming-up-a-java-process)
* [Running Database Migrations for Java Apps](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps)
* [Reducing the Slug Size of Java Applications](https://devcenter.heroku.com/articles/reducing-the-slug-size-of-java-applications)

### [Keep reading](https://devcenter.heroku.com/articles/java-support#keep-reading)

* [Java](https://devcenter.heroku.com/categories/java-support)

### [Feedback](https://devcenter.heroku.com/articles/java-support#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fjava-support&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Java Session Handling on Heroku

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/java-session-handling-on-heroku)

Last updated July 15, 2022

## Table of Contents

* [Why Use Redis to Store Sessions?](https://devcenter.heroku.com/articles/java-session-handling-on-heroku#why-use-redis-to-store-sessions)
* [Storing Sessions with Tomcat Webapp Runner](https://devcenter.heroku.com/articles/java-session-handling-on-heroku#storing-sessions-with-tomcat-webapp-runner)
* [Storing Sessions with Redisson](https://devcenter.heroku.com/articles/java-session-handling-on-heroku#storing-sessions-with-redisson)
* [Other Options](https://devcenter.heroku.com/articles/java-session-handling-on-heroku#other-options)

HTTP is a stateless protocol, but most applications need to preserve certain information across requests, such as login state or the contents of a shopping cart. This kind of state is usually stored in a [session](https://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpSession.html).

You can either store sessions on the client side in encrypted HTTP cookies or on the server side with a variety of persistence mechanisms. Both approaches have advantages and disadvantages, but this article describes how to reliably handle server-side sessions in Java applications on Heroku. All of the examples in this article use [Redis](https://devcenter.heroku.com/articles/heroku-redis) as the storage mechanism.

## [Why Use Redis to Store Sessions?](https://devcenter.heroku.com/articles/java-session-handling-on-heroku#why-use-redis-to-store-sessions)

File-based session storage and in-memory storage are discouraged on Heroku because of the stateless nature of [dynos](https://devcenter.heroku.com/articles/dynos) and the need to scale them horizontally. In both file and in-memory storage, session data is lost on restart and cannot be shared between dynos. Furthermore, if sessions are stored local to a dyno, the next request from a user might end up on a different dyno, where the session information does not exist.

A Redis instance persists data across dyno restarts and can be shared between multiple dynos, which makes it an excellent solution for storing sessions. You can add a Redis instance to your app by running:

heroku addons:create heroku-redis

This populates a REDIS\_URL config var on your app, which is ready to use with the strategies described in this article. Alternatively, you can use any of Heroku’s [third-party Redis add-ons](https://elements.heroku.com/search/addons?q=redis).

## [Storing Sessions with Tomcat Webapp Runner](https://devcenter.heroku.com/articles/java-session-handling-on-heroku#storing-sessions-with-tomcat-webapp-runner)

Heroku’s recommended servlet container, Tomcat [Webapp Runner](https://github.com/jsimone/webapp-runner), has built-in support for session storage with either Redis or Memcached. To use it with the Heroku Data for Redis add-on, pass the --session-store redis option to your Webapp Runner JAR file by adding it to your Procfile like this:

web: java -jar target/dependency/webapp-runner.jar --session-store redis target/myapp.war

Alternatively, if you’re using either the heroku war:deploy or mvn heroku:deploy-war commands (and therefore do not have a Procfile), you can set this option as a config var by running:

heroku config:set WEBAPP\_RUNNER\_OPTS="--session-store redis"

Webapp Runner detects the REDIS\_URL environment variable provided by the add-on and starts using it to store your application’s HTTP sessions.

## [Storing Sessions with Redisson](https://devcenter.heroku.com/articles/java-session-handling-on-heroku#storing-sessions-with-redisson)

If you’re not using Webapp Runner, then you need a library like [Redisson](https://github.com/redisson/redisson), a popular, high-performance Java client for Redis. It can be used for many different purposes, including session storage.

Redisson provides adapters for [Tomcat](https://github.com/redisson/redisson/wiki/14.-Integration%20with%20frameworks#145-tomcat-redis-session-manager) and [Spring](https://github.com/redisson/redisson/wiki/14.-Integration%20with%20frameworks#146-spring-session). For other servers and frameworks, you need to use the Redisson API directly.

In all cases, you must create an org.redisson.config.Config object in your Java code like this:

String redisUriString = System.getenv("REDIS\_URL");

URI redisUri = URI.create(redisUriString);

Config config = new Config();

SingleServerConfig serverConfig = config.useSingleServer()

.setAddress(redisUriString)

.setConnectionPoolSize(10)

.setConnectionMinimumIdleSize(10)

.setTimeout(5000);

if (redisUri.getUserInfo() != null) {

serverConfig.setPassword(redisUri.getUserInfo().substring(redisUri.getUserInfo().indexOf(":")+1));

}

Then you can create a new RedissonClient instance using the Config object:

RedissonClient redisson = Redisson.create(config);

If you’re using Spring or Spring Boot, you also need the @EnableRedissonHttpSession annotation, as described in the [Redisson documentation](https://github.com/redisson/redisson/wiki/14.-Integration%20with%20frameworks#146-spring-session).

How you store objects with the client depends on how your framework or server works. The [Redisson documentation has a number of examples and common patterns](https://github.com/redisson/redisson).

## [Other Options](https://devcenter.heroku.com/articles/java-session-handling-on-heroku#other-options)

There are many other clients and libraries that facilitate session store in both Redis and Memcached. They include:

* [Jedis](https://github.com/xetorthio/jedis): a Java Redis client
* [XMemcached](https://github.com/killme2008/xmemcached): a Java Memcached client

You can also check the [add-on provider documentation for Memcachier](https://devcenter.heroku.com/articles/memcachier#java) for examples of connecting to Memcached.

### [Keep reading](https://devcenter.heroku.com/articles/java-session-handling-on-heroku#keep-reading)

* [Java](https://devcenter.heroku.com/categories/java-support)

### [Feedback](https://devcenter.heroku.com/articles/java-session-handling-on-heroku#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fjava-session-handling-on-heroku&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Setting the HTTP Port for Java Applications

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/setting-the-http-port-for-java-applications)

Last updated April 09, 2020

## Table of Contents

* [Spring Boot](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#spring-boot)
* [Webapp Runner (Tomcat)](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#webapp-runner-tomcat)
* [Play Framework](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#play-framework)
* [Dropwizard](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#dropwizard)
* [Thorntail](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#thorntail)
* [Ratpack](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#ratpack)
* [Grails](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#grails)

Heroku expects a web application to bind its HTTP server to the port defined by the $PORT environment variable. Many frameworks default to port 8080, but can be configured to use an environment variable instead. In most cases, the port can be configured by adding a parameter to the java command in an application’s Procfile, but some frameworks provide a configuration file. The most common Java frameworks are listed here.

## [Spring Boot](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#spring-boot)

Spring-Boot provides a few different mechanisms for setting the HTTP port. It can be passed an option to the executable JAR file (i.e. the options after the -jar app.jar options in your java command):

--server.port=$PORT

Or as a Java system property:

-Dserver.port=$PORT

In both cases, these options can be added to the java command in an app’s Procfile. Another option is a configuration element in the application.yml:

server:

port: $PORT

For more info see [Spring Boot documentation on Properties and Configuration](https://docs.spring.io/spring-boot/docs/current/reference/html/howto-properties-and-configuration.html)

## [Webapp Runner (Tomcat)](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#webapp-runner-tomcat)

[Webapp Runner](https://github.com/jsimone/webapp-runner) allows you to launch an exploded or compressed war on your filesystem into a [Tomcat](https://tomcat.apache.org/) container with a simple java -jar command. It accepts the following option to the executable JAR file (i.e. the options after the -jar webapp-runner.jar options in your java command):

--port $PORT

When using the heroku war:deploy and Heroku Maven Plugin tools, this option is configured for you. For more information, see the [Webapp Runner documentation](https://github.com/jsimone/webapp-runner/blob/master/README.md).

## [Play Framework](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#play-framework)

The [Play Framework](https://playframework.com/), which uses [Netty](https://netty.io/) as its server, accepts a Java system property (i.e. an option to the java command):

-Dhttp.port=$PORT

For more info see the [Play documentation for Heroku](https://www.playframework.com/documentation/2.6.x/ProductionHeroku).

## [Dropwizard](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#dropwizard)

The Dropwizard framework accepts a Java system property:

-Ddw.server.applicationConnectors[0].port=$PORT

For more information see the [Dropwizard Configuration reference](https://www.dropwizard.io/en/latest/manual/core.html#environment-variables).

## [Thorntail](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#thorntail)

The [Thorntail](https://thorntail.io/) framework accepts a system property:

-Dswarm.http.port=$PORT

For more information see the [WildFly Swarm documentation](https://docs.thorntail.io/2018.3.3/).

## [Ratpack](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#ratpack)

The [Ratpack](https://ratpack.io/) framework, which uses the [Netty](https://netty.io/) server, automatically detects the $PORT environment variable and configures the app to use it.

## [Grails](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#grails)

The Grails 3 framework produces a WAR file that can be run with either Tomcat or Jetty. See the [Webapp Runner section](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications) for more information.

For more information see the [WildFly Swarm](https://docs.thorntail.io/2018.3.3/#configuring-a-wildfly-swarm-application)

### [Keep reading](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#keep-reading)

* [Java](https://devcenter.heroku.com/categories/java-support)

### [Feedback](https://devcenter.heroku.com/articles/setting-the-http-port-for-java-applications#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fsetting-the-http-port-for-java-applications&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Using a Custom Maven Settings File

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/using-a-custom-maven-settings-xml)

Last updated December 16, 2019

## Table of Contents

* [Creating a custom settings file](https://devcenter.heroku.com/articles/using-a-custom-maven-settings-xml#creating-a-custom-settings-file)
* [Defining the MAVEN\_SETTINGS\_PATH config variable](https://devcenter.heroku.com/articles/using-a-custom-maven-settings-xml#defining-the-maven_settings_path-config-variable)
* [Defining the MAVEN\_SETTINGS\_URL config variable](https://devcenter.heroku.com/articles/using-a-custom-maven-settings-xml#defining-the-maven_settings_url-config-variable)
* [Using password protected repositories](https://devcenter.heroku.com/articles/using-a-custom-maven-settings-xml#using-password-protected-repositories)

A Maven settings.xml file defines values that [configure Maven execution in various ways](https://maven.apache.org/settings.html). Most commonly, it is used to define a local repository location, alternate remote repository servers, and authentication information for private repositories. In this article you’ll learn how to add a custom setting file to a Java application deployed on Heroku.

If you already have a Java application, you may use it for this example. Otherwise, create a simple application from the [Getting Started with Java on Heroku](https://devcenter.heroku.com/articles/getting-started-with-java) article.

## [Creating a custom settings file](https://devcenter.heroku.com/articles/using-a-custom-maven-settings-xml#creating-a-custom-settings-file)

When a file named settings.xml is present in the root directory of an application, [Heroku’s Java Support](https://devcenter.heroku.com/articles/java-support) will automatically use it to configure Maven at compile time.

To demonstrate this, add a settings.xml file to the root directory of your Java project and put the following code in it.

<?xml version="1.0" encoding="UTF-8"?>

<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">

<profiles>

<profile>

<id>jboss-public</id>

<repositories>

<repository>

<id>jboss-public-repository</id>

<name>JBoss Public Maven Repository Group</name>

<url>http://repository.jboss.org/nexus/content/groups/public/</url>

</repository>

</repositories>

</profile>

</profiles>

<activeProfiles>

<activeProfile>jboss-public</activeProfile>

</activeProfiles>

</settings>

This tells Maven to search the repository hosted at <http://repository.jboss.org/> when resolving dependencies for your application. The repository must be addressed with HTTP or HTTPS. If you require a file:// address, then see the [Unmanaged Dependencies article](https://devcenter.heroku.com/articles/local-maven-dependencies).

We can test the settings locally by adding the -s option to any Maven task. But first, we’ll need to add a new dependency to the project. The jboss.web.servlet-api library is a good example because it’s only available on the JBoss repository. Add the follow element to your project’s pom.xml:

<dependency>

<groupId>jboss.web</groupId>

<artifactId>servlet-api</artifactId>

<version>2.1.0.GA</version>

</dependency>

Now run the following command and Maven will download the artifact.

mvn -s settings.xml dependency:list

[INFO] Scanning for projects...

...

Downloading: http://repository.jboss.org/nexus/content/groups/public/jboss/web/servlet-api/2.1.0.GA/servlet-api-2.1.0.GA.pom

Downloaded: http://repository.jboss.org/nexus/content/groups/public/jboss/web/servlet-api/2.1.0.GA/servlet-api-2.1.0.GA.pom (195 B at 0.2 KB/sec)

Downloading: http://repository.jboss.org/nexus/content/groups/public/jboss/web/servlet-api/2.1.0.GA/servlet-api-2.1.0.GA.jar

Downloaded: http://repository.jboss.org/nexus/content/groups/public/jboss/web/servlet-api/2.1.0.GA/servlet-api-2.1.0.GA.jar (84 KB at 90.8 KB/sec)

If we had not used the custom settings file, the build would have failed with Maven producing an error like this:

Downloading: http://repo.maven.apache.org/maven2/jboss/web/servlet-api/2.1.0.GA/servlet-api-2.1.0.GA.pom

[WARNING] The POM for jboss.web:servlet-api:jar:2.1.0.GA is missing, no dependency information available

Downloading: http://repo.maven.apache.org/maven2/jboss/web/servlet-api/2.1.0.GA/servlet-api-2.1.0.GA.jar

...

[ERROR] Failed to execute goal on project helloworld: Could not resolve dependencies for project com.example:helloworld:jar:1.0-SNAPSHOT: Could not find artifact jboss.web:servlet-api:jar:2.1.0.GA in central (http://repo.maven.apache.org/maven2) -> [Help 1]

Now add the settings.xml and the pom.xml changes to your Git repository and deploy to Heroku like so:

git add pom.xml settings.xml

git commit -m "adding jboss servlet-api dependency"

git push heroku master

As Maven runs on the dyno, we see the same output we saw locally. This works because Heroku detects the settings.xml file in the root directory, and adds the -s option to the Maven command. Next, we’ll discuss how to customize this location.

## [Defining the MAVEN\_SETTINGS\_PATH config variable](https://devcenter.heroku.com/articles/using-a-custom-maven-settings-xml#defining-the-maven_settings_path-config-variable)

If you do not want the settings.xml file in the root directory or if you intend to frequently change between different setting configurations, you may prefer to put a settings file in a custom location. Heroku provides this capability with the MAVEN\_SETTINGS\_PATH config variable.

We can demonstrate this feature by moving the existing settings.xml into a support/ directory and renaming it like so:

mkdir -p support

git mv settings.xml support/jboss-settings.xml

Now we tell Heroku where the settings file is located by defining MAVEN\_SETTINGS\_PATH relative to the root directory.

heroku config:set MAVEN\_SETTINGS\_PATH=support/jboss-settings.xml

Before testing the change, we increment the version of servlet-api – forcing Maven to download it again. Change the dependency in the pom.xml to version 2.1.1.GA. It should look like this:

<dependency>

<groupId>jboss.web</groupId>

<artifactId>servlet-api</artifactId>

<version>2.1.1.GA</version>

</dependency>

Now commit the changes to Git, and redeploy to Heroku like so:

git commit -am "moved settings file and incremented servlet-api version"

git push heroku master

Once again, Maven will download the dependency from the JBoss repository. This provides a bit of flexibility, but sometimes a custom location in your project is not enough. You may want to keep the file out of your code base altogether.

## [Defining the MAVEN\_SETTINGS\_URL config variable](https://devcenter.heroku.com/articles/using-a-custom-maven-settings-xml#defining-the-maven_settings_url-config-variable)

When the MAVEN\_SETTINGS\_URL config variable is defined, Heroku will download the file at the given location and use it to configure Maven. Before demonstrating this, we must unset the variable we defined in the previous example because it will take precedence if both variables are set:

heroku config:unset MAVEN\_SETTINGS\_PATH

Now we can use a settings.xml from a publicly available source such as the Torquebox Application Server code-base. Set the config variable like this:

heroku config:set MAVEN\_SETTINGS\_URL="https://raw.githubusercontent.com/torquebox/torquebox/master/support/settings.xml"

As before, increment the servlet-api version to 2.1.2.GA, add the pom.xml to the Git repo, commit the changes and redeploy to Heroku. Maven will download the new artifact from the JBoss repository as it did earlier.

The JBoss repository is convenient because it is public – it does not require a password to access. But not all repositories are quite so open.

## [Using password protected repositories](https://devcenter.heroku.com/articles/using-a-custom-maven-settings-xml#using-password-protected-repositories)

Some artifact repositories require a username and password to access. Many times, the repository is a private server hosting internal artifacts. When this is the case, credentials for accessing the repository must be provided in the settings.xml, which can be a problem if the file is checked into a Git repository.

Fortunately, Maven settings files can detect environment variables. Any token in the form ${env.ENV\_VAR} (where ENV\_VAR is the name of a variable) will resolve to the value of the associated environment variable. Thus we can define the password for a private Maven repo as a Heroku config variable like this:

heroku config:set MAVEN\_REPO\_PASSWORD="deployment123"

Then we can use the variable in our settings.xml file by creating a <server> element that matches the <id> of the <repository> element in the <activeProfile>.

<servers>

<server>

<id>my-private-repo</id>

<username>deployment</username>

<password>${env.MAVEN\_REPO\_PASSWORD}</password>

</server>

</servers>

In order to demonstrate this, you will need a private Maven repository. You can create a repository by downloading [Sonatype Nexus](http://www.sonatype.org/nexus/) and following the [instructions for securing a repository](http://books.sonatype.com/nexus-book/reference/index.html). Or you may use a hosted version of [JFrog Artifactory](http://www.jfrog.com/home/v_artifactorycloud_overview).

In either case a private repository can be used to publish internal artifacts and include them in your Heroku applications.

For more information on customizing Maven with a settings file, please see the [Maven Settings Reference](https://maven.apache.org/settings.html) from Apache.org.

### [Keep reading](https://devcenter.heroku.com/articles/using-a-custom-maven-settings-xml#keep-reading)

* [Working with Maven](https://devcenter.heroku.com/categories/working-with-maven)

### [Feedback](https://devcenter.heroku.com/articles/using-a-custom-maven-settings-xml#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fusing-a-custom-maven-settings-xml&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Adding Unmanaged Dependencies to a Maven Project

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/local-maven-dependencies)

Last updated December 16, 2019

## Table of Contents

* [Pick groupId, artifactId and version parameters](https://devcenter.heroku.com/articles/local-maven-dependencies#pick-groupid-artifactid-and-version-parameters)
* [Create a local Maven repository directory](https://devcenter.heroku.com/articles/local-maven-dependencies#create-a-local-maven-repository-directory)
* [Deploy the Artifact Into the Repo](https://devcenter.heroku.com/articles/local-maven-dependencies#deploy-the-artifact-into-the-repo)
* [Update Pom file](https://devcenter.heroku.com/articles/local-maven-dependencies#update-pom-file)
* [Commit to Git](https://devcenter.heroku.com/articles/local-maven-dependencies#commit-to-git)

Some Java applications have dependencies that aren’t available in a public or private Maven repository (the latter of which can be accessed using a [custom settings.xml file](https://devcenter.heroku.com/articles/using-a-custom-maven-settings-xml)). This guide shows you how to add these unmanaged libraries in your application project and tell Maven how to find them.

## [Pick groupId, artifactId and version parameters](https://devcenter.heroku.com/articles/local-maven-dependencies#pick-groupid-artifactid-and-version-parameters)

Let’s say your app depends on the library mylib.jar which is not in any public Maven repository. First you must define a groupId, artifactId and version for the library. These parameters may not matter to you, but Maven requires this information for all dependencies.

So let’s use:

* groupId: com.example
* artifactId: mylib
* version: 1.0 (or whatever version your lib is, if you have it versioned)

## [Create a local Maven repository directory](https://devcenter.heroku.com/articles/local-maven-dependencies#create-a-local-maven-repository-directory)

Your project root should look something like this to start with:

yourproject

+- pom.xml

+- src

Add a standard Maven repository directory called repo for the group com.example and version 1.0:

yourproject

+- pom.xml

+- src

+- repo

## [Deploy the Artifact Into the Repo](https://devcenter.heroku.com/articles/local-maven-dependencies#deploy-the-artifact-into-the-repo)

Maven can deploy the artifact for you using the mvn deploy:deploy-file goal:

mvn deploy:deploy-file -Durl=file:///path/to/yourproject/repo/ -Dfile=mylib-1.0.jar -DgroupId=com.example -DartifactId=mylib -Dpackaging=jar -Dversion=1.0

Your project will contain some Maven metadata and your JAR file.

yourproject

+- pom.xml

+- src

+- repo

+- com

+- example

+- mylib

+- maven-metadata.xml

+- ...

+- 1.0

+- mylib-1.0.jar

+- mylib-1.0.pom

+- ...

## [Update Pom file](https://devcenter.heroku.com/articles/local-maven-dependencies#update-pom-file)

Now edit your pom.xml and add this repository to your <repositories/> element. You may have to create the <repositories/> element if you dont have one.

<repositories>

<!--other repositories if any-->

<repository>

<id>project.local</id>

<name>project</name>

<url>file:${project.basedir}/repo</url>

</repository>

</repositories>

When using the repository from a sub-module, you will need to substitute the ${project.parent.baseDir} property in the <url> element.

Now you can add this jar as a dependency as normal:

<dependency>

<groupId>com.example</groupId>

<artifactId>mylib</artifactId>

<version>1.0</version>

</dependency>

## [Commit to Git](https://devcenter.heroku.com/articles/local-maven-dependencies#commit-to-git)

Dont forget to add and commit your local repo folder to git.

$ git add repo

$ git commit -m 'adding project local maven repo, with mylib.jar 1.0'

The next time you push your project the dependency will resolve and your application will build without an issue. Since the local repo folder and jars are checked in with your application code they will remain private to your app.

### [Keep reading](https://devcenter.heroku.com/articles/local-maven-dependencies#keep-reading)

* [Working with Maven](https://devcenter.heroku.com/categories/working-with-maven)

### [Feedback](https://devcenter.heroku.com/articles/local-maven-dependencies#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Flocal-maven-dependencies&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Deploying Java Applications with the Heroku Maven Plugin

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/deploying-java-applications-with-the-heroku-maven-plugin)

Last updated February 06, 2023

## Table of Contents

* [Adding the plugin](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin#adding-the-plugin)
* [Deploying with the plugin](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin#deploying-with-the-plugin)
* [Advanced configuration](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin#advanced-configuration)
* [Deploying to multiple applications](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin#deploying-to-multiple-applications)

In addition to [Git deployment](https://devcenter.heroku.com/articles/git), Heroku supports [building and releasing apps via an API](https://devcenter.heroku.com/articles/build-and-release-using-the-api). The [Heroku Maven plugin](https://github.com/heroku/heroku-maven-plugin) utilizes this API to provide direct deployment of prepackaged standalone web applications to Heroku.

This may be a preferred approach for applications that take a long time to compile, or that need to be deployed from a Continuous Integration server such as [Travis CI](http://travis-ci.org) or Jenkins.

In this article, you’ll learn how to include the Heroku Maven Plugin in your project, configure it, and deploy your application to Heroku.

## [Adding the plugin](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin#adding-the-plugin)

To include the plugin in your project, add the following to your pom.xml file:

<project>

...

<build>

...

<plugins>

<plugin>

<groupId>com.heroku.sdk</groupId>

<artifactId>heroku-maven-plugin</artifactId>

<version>3.0.7</version>

</plugin>

</plugins>

</build>

</project>

Then create a new Heroku application by running:

heroku create

If your application is under version control with Git, then the plugin will detect the application name by default. If you are not using Git, you must add this configuration to the plugin (but replace the app name with the name of your app):

<configuration>

<appName>sushi</appName>

</configuration>

If your application is packaged as a WAR file, then this configuration is sufficient, but if your application is a standalone Java application then you’ll need to define a process type.

A process type is a command used to start your application. You can configure these with a Procfile as describe in the [Process Types and the Procfile](https://devcenter.heroku.com/articles/procfile) article. Or you can configure them directly in your pom.xml like this:

<configuration>

...

<processTypes>

<web>java $JAVA\_OPTS -cp target/classes:target/dependency/\* Main</web>

</processTypes>

</configuration>

The configuration above tells Heroku to start a web process using the Main class. Your process type will differ depending on your application code.

Now you’re ready to deploy.

## [Deploying with the plugin](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin#deploying-with-the-plugin)

If your application is packaged as a WAR, then you can create the slug and deploy it to Heroku with the following command:

mvn clean heroku:deploy-war

...

[INFO] -----> Packaging application...

[INFO] - app: obscure-sierra-7788

[INFO] - including: ./target/dependency/webapp-runner.jar

[INFO] - including: ./target/my-app.war

[INFO] -----> Creating build...

[INFO] - file: target/heroku/slug.tgz

[INFO] - size: 1MB

[INFO] -----> Uploading build...

[INFO] - success

[INFO] -----> Deploying...

[INFO] remote:

[INFO] remote: -----> Fetching custom tar buildpack... done

[INFO] remote: -----> JVM Common app detected

[INFO] remote: -----> Installing OpenJDK 1.8... done

[INFO] remote: -----> Discovering process types

[INFO] remote: Procfile declares types -> web

[INFO] remote:

[INFO] remote: -----> Compressing... done, 49.5MB

[INFO] remote: -----> Launching... done, v157

[INFO] remote: https://obscure-sierra-7788.herokuapp.com/ deployed to Heroku

[INFO] remote:

[INFO] -----> Done

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 01:04 min

[INFO] Finished at: 2014-11-11T16:07:11-06:00

[INFO] Final Memory: 20M/304M

[INFO] ------------------------------------------------------------------------

If your application is standalone (and thus required a process type), you can deploy with this command:

mvn clean heroku:deploy

...

[INFO] -----> Packaging application...

[INFO] - app: obscure-sierra-7788

[INFO] - including: ./target/

[INFO] -----> Creating build...

[INFO] - file: target/heroku/slug.tgz

[INFO] - size: 1MB

[INFO] -----> Uploading build...

[INFO] - success

[INFO] -----> Deploying...

[INFO] remote:

[INFO] remote: -----> Fetching custom tar buildpack... done

[INFO] remote: -----> JVM Common app detected

[INFO] remote: -----> Installing OpenJDK 1.8... done

[INFO] remote: -----> Discovering process types

[INFO] remote: Procfile declares types -> web

[INFO] remote:

[INFO] remote: -----> Compressing... done, 49.5MB

[INFO] remote: -----> Launching... done, v157

[INFO] remote: https://obscure-sierra-7788.herokuapp.com/ deployed to Heroku

[INFO] remote:

[INFO] -----> Done

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 01:04 min

[INFO] Finished at: 2014-11-11T16:07:11-06:00

[INFO] Final Memory: 20M/304M

[INFO] ------------------------------------------------------------------------

Now you can visit your application with this command:

heroku open

Or view the logs with this command:

heroku logs

The Heroku CLI will allow you to access the application, and run commands just like an application deployed with git push.

## [Advanced configuration](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin#advanced-configuration)

The Maven plugin allows for several advanced configuration settings. All of these settings are defined in the <configuration> element for the plugin in the pom.xml file. For example, you may set the desired JDK version like so:

<jdkVersion>1.8</jdkVersion>

For valid values and the current default version, see [Heroku’s Java support](https://devcenter.heroku.com/articles/java-support#specifying-a-java-version).

You can set configuration variables:

<configVars>

<MY\_VAR>SomeValue</MY\_VAR>

<JAVA\_OPTS>-Xss512k -XX:+UseCompressedOops</JAVA\_OPTS>

</configVars>

Be aware that any variable defined in <configVars> will override defaults, or previously defined config vars.

You can also include additional directories in the slug like this (note that they must be relative to the project root):

<includes>

<include>etc/readme.txt</include>

</includes>

By default, the plugin will package the essential directories under your project’s target directory, so including additional directories should not be necessary in most cases.

You can specify an alternate location for your WAR file like this:

<warFile>relative/path/myapp.war</warFile>

But this is only accepted when deploying with the heroku:deploy-war goal.

## [Deploying to multiple applications](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin#deploying-to-multiple-applications)

Most real applications will be required to deploy to multiple Heroku apps (such as dev, test, and prod). Normally this is done with multiple Git remotes. But with the plugin you can use either system properties, environment variables, or any other native Maven/Java configuration method. For example, you might define your heroku.appName as a system property like this:

mvn heroku:deploy -Dheroku.appName=myapp

However, the preferred approach uses Maven profiles. An example configuration might look like this:

<build>

<plugins>

<plugin>

<groupId>com.heroku.sdk</groupId>

<artifactId>heroku-maven-plugin</artifactId>

<configuration>

<processTypes>

<web>java $JAVA\_OPTS -cp target/classes:target/dependency/\* Main</web>

</processTypes>

</configuration>

</plugin>

</plugins>

</build>

<profiles>

<profile>

<id>test</id>

<build>

<plugins>

<plugin>

<groupId>com.heroku.sdk</groupId>

<artifactId>heroku-maven-plugin</artifactId>

<configuration>

<appName>myapp-test</appName>

</configuration>

</plugin>

</plugins>

</build>

</profile>

<profile>

<id>prod</id>

<build>

<plugins>

<plugin>

<groupId>com.heroku.sdk</groupId>

<artifactId>heroku-maven-plugin</artifactId>

<configuration>

<appName>myapp-prod</appName>

</configuration>

</plugin>

</plugins>

</build>

</profile>

</profiles>

For more information on the plugin and Java deployment in general, see [Heroku’s Java support](https://devcenter.heroku.com/articles/java-support). For more information on Maven, see the [Apache Maven documentation](https://maven.apache.org/).

### [Keep reading](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin#keep-reading)

* [Working with Maven](https://devcenter.heroku.com/categories/working-with-maven)

### [Feedback](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fdeploying-java-applications-with-the-heroku-maven-plugin&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Connecting to Relational Databases on Heroku with Java

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/connecting-to-relational-databases-on-heroku-with-java)

Last updated November 14, 2022

## Table of Contents

* [Using the JDBC\_DATABASE\_URL](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-jdbc_database_url)
* [Using the SPRING\_DATASOURCE\_URL in a Spring Boot app](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-spring_datasource_url-in-a-spring-boot-app)
* [Using the JDBC\_DATABASE\_URL in a Spring Boot app](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-jdbc_database_url-in-a-spring-boot-app)
* [Using the DATABASE\_URL in a Play Framework app](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-database_url-in-a-play-framework-app)
* [Using the DATABASE\_URL in plain JDBC](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-database_url-in-plain-jdbc)
* [Using the DATABASE\_URL in Spring with XML configuration](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-database_url-in-spring-with-xml-configuration)
* [Using the DATABASE\_URL in Spring with Java configuration](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-database_url-in-spring-with-java-configuration)
* [Using the DATABASE\_URL with Hibernate](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-database_url-with-hibernate)
* [Using SSL with PostgreSQL](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-ssl-with-postgresql)
* [Connecting to a database remotely](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#connecting-to-a-database-remotely)
* [Running database migrations](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#running-database-migrations)
* [Using with Heroku Postgres Connection Pooling](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-with-heroku-postgres-connection-pooling)
* [Sample project](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#sample-project)
* [Further learning](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#further-learning)

Applications on Heroku can use a variety of relational database services including the [Postgres database](https://devcenter.heroku.com/categories/heroku-postgres) offered by Heroku.

Databases are provisioned using the add-on system. Some applications provisions a Heroku Postgres database by default. You can check if you have one by running heroku info and looking at the list of add-ons:

heroku info

=== sparkling-wine-2003

Web URL: http://sparkling-wine-2003.herokuapp.com/

Git Repo: git@heroku.com:sparkling-wine-2003.git

Repo size: 21M

Slug size: 916k

Stack: heroku-18

Data size: (empty)

Addons: Heroku Postgresql Mini

Owner: jesper@heroku.com

It depends on the buildpack whether a database is provisioned automatically. You can provision a database manually with

heroku addons:create heroku-postgresql

After provisioning a relational database to your application. The application reads the database connection information from the DATABASE\_URL config variable. It is formatted like this:

[database type]://[username]:[password]@[host]:[port]/[database name]

For instance:

postgres://foo:foo@heroku.com:5432/hellodb

You can see the DATABASE\_URL provided to an application by running:

heroku config

DATABASE\_URL => postgres://foo:foo@heroku.com:5432/hellodb

It is not recommended to copy this value into a static file since the environment may change the value. Instead an application should read the DATABASE\_URL environment variable (or JDBC\_DATABASE\_URL variable described later) and set up the database connections based on that information.

## [Using the JDBC\_DATABASE\_URL](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-jdbc_database_url)

The official Heroku buildpacks for Java, Scala, Clojure, and Gradle will attempt to create a JDBC\_DATABASE\_URL environment variable when a dyno starts up. This variable is dynamic and will not appear in your list of configuration variables when running heroku config. You can view it by running the following command:

heroku run echo \$JDBC\_DATABASE\_URL

On Windows, you must omit the leading `` or just run `heroku run env`.

The variable will include ?user=<user>&password=<password> parameters, but JDBC\_DATABASE\_USERNAME and JDBC\_DATABASE\_PASSWORD environment variables will also be set when possible.

The authoritative source of the Database URL is still the DATABASE\_URL environment variable, but the JDBC\_DATABASE\_URL can be used in most cases.

The Java buildpack will also set the JDBC\_DATABASE\_URL for the [ClearDB MySQL](https://elements.heroku.com/addons/cleardb), [JawsDB MySQL](https://elements.heroku.com/addons/jawsdb), and [JawsDB Maria](https://elements.heroku.com/addons/jawsdb-maria) add-ons.

For databases using the HEROKU\_POSTGRESQL\_<COLOR> format, the Java buildpack will create HEROKU\_POSTGRESQL\_<COLOR>\_JDBC\_URL, HEROKU\_POSTGRESQL\_<COLOR>\_JDBC\_USERNAME, and HEROKU\_POSTGRESQL\_<COLOR>\_JDBC\_PASSWORD config variables.

## [Using the SPRING\_DATASOURCE\_URL in a Spring Boot app](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-spring_datasource_url-in-a-spring-boot-app)

The official Heroku buildpacks for Java and Gradle will attempt to create SPRING\_DATASOURCE\_URL, SPRING\_DATASOURCE\_USERNAME, and SPRING\_DATASOURCE\_PASSWORD environment variables when a dyno starts up. The values of the variables will be identical to the values in the corresponding JDBC variables.

As long as your application has the proper JDBC driver defined as a dependency, these environment variables should allow your Spring Boot application to connect to the database without any other configuration.

If you need to override the predefined SPRING\_DATASOURCE\_\* environment variables you can set them yourself with the heroku config:set command or in set them in the dashboard. Alternatively, you can add the -Dspring.datasource.url property to your Procfile, which will [take precedence over the OS-level environment variables](http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html).

## [Using the JDBC\_DATABASE\_URL in a Spring Boot app](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-jdbc_database_url-in-a-spring-boot-app)

Spring Boot allows you to externalize your configuration so you can work with the same application code in different environments. You can use properties files, YAML files, environment variables and command-line arguments to externalize configuration.

You can set your database URL in an application.yml file like this:

spring:

datasource:

url: ${JDBC\_DATABASE\_URL}

username: ${JDBC\_DATABASE\_USERNAME}

password: ${JDBC\_DATABASE\_PASSWORD}

For more information see the official Spring documentation on [Externalized Configuration](http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html).

## [Using the DATABASE\_URL in a Play Framework app](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-database_url-in-a-play-framework-app)

Play Framework supports the DATABASE\_URL environment variable out-of-the box. The built-in ORM framework can be configured to use this value by adding it to your conf/application.conf:

db.default=${DATABASE\_URL}

When using Slick with the Play Framework, the configuration looks like this:

slick.dbs.default.driver="slick.driver.PostgresDriver$"

slick.dbs.default.db.dataSourceClass = "slick.jdbc.DatabaseUrlDataSource"

slick.dbs.default.db.properties.driver = "org.postgresql.Driver"

For more information see the Play documentation for the [Play Slick module](https://www.playframework.com/documentation/2.4.x/PlaySlick).

## [Using the DATABASE\_URL in plain JDBC](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-database_url-in-plain-jdbc)

To instantiate a JDBC connection in your code, you can use a method like this:

private static Connection getConnection() throws URISyntaxException, SQLException {

String dbUrl = System.getenv("JDBC\_DATABASE\_URL");

return DriverManager.getConnection(dbUrl);

}

When using DATABASE\_URL directly, it would look like this:

private static Connection getConnection() throws URISyntaxException, SQLException {

URI dbUri = new URI(System.getenv("DATABASE\_URL"));

String username = dbUri.getUserInfo().split(":")[0];

String password = dbUri.getUserInfo().split(":")[1];

String dbUrl = "jdbc:postgresql://" + dbUri.getHost() + ':' + dbUri.getPort() + dbUri.getPath();

return DriverManager.getConnection(dbUrl, username, password);

}

**Note**:

The DATABASE\_URL for the Heroku Postgres add-on follows the below convention

postgres://<username>:<password>@<host>:<port>/<dbname>

However the Postgres JDBC driver uses the following convention:

jdbc:postgresql://<host>:<port>/<dbname>?user=<username>&password=<password>

Notice the additional ql at the end of the URL scheme. Due to this difference, for Postgres, you may need to hardcode the scheme to postgresql in your Java class or your Spring XML configuration.

## [Using the DATABASE\_URL in Spring with XML configuration](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-database_url-in-spring-with-xml-configuration)

This snippet of Spring XML configuration will setup a BasicDataSource from the DATABASE\_URL and can then be used with Hibernate, JPA, etc:

<bean class="java.net.URI" id="dbUrl">

<constructor-arg value="#{systemEnvironment['DATABASE\_URL']}"/>

</bean>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">

<property name="url" value="#{ 'jdbc:postgresql://' + @dbUrl.getHost() + ':' + @dbUrl.getPort() + @dbUrl.getPath() }"/>

<property name="username" value="#{ @dbUrl.getUserInfo().split(':')[0] }"/>

<property name="password" value="#{ @dbUrl.getUserInfo().split(':')[1] }"/>

</bean>

## [Using the DATABASE\_URL in Spring with Java configuration](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-database_url-in-spring-with-java-configuration)

Alternatively you can use Java for configuration of the BasicDataSource in Spring:

@Configuration

public class MainConfig {

@Bean

public BasicDataSource dataSource() throws URISyntaxException {

String dbUrl = System.getenv("JDBC\_DATABASE\_URL");

String username = System.getenv("JDBC\_DATABASE\_USERNAME");

String password = System.getenv("JDBC\_DATABASE\_PASSWORD");

BasicDataSource basicDataSource = new BasicDataSource();

basicDataSource.setUrl(dbUrl);

basicDataSource.setUsername(username);

basicDataSource.setPassword(password);

return basicDataSource;

}

}

When using DATABASE\_URL directly, it would look like this:

@Configuration

public class MainConfig {

@Bean

public BasicDataSource dataSource() throws URISyntaxException {

URI dbUri = new URI(System.getenv("DATABASE\_URL"));

String username = dbUri.getUserInfo().split(":")[0];

String password = dbUri.getUserInfo().split(":")[1];

String dbUrl = "jdbc:postgresql://" + dbUri.getHost() + ':' + dbUri.getPort() + dbUri.getPath();

BasicDataSource basicDataSource = new BasicDataSource();

basicDataSource.setUrl(dbUrl);

basicDataSource.setUsername(username);

basicDataSource.setPassword(password);

return basicDataSource;

}

}

Again, note the difference in the sub-protocol. The DATABASE\_URL uses postgres while JDBC requires the value postgresql.

## [Using the DATABASE\_URL with Hibernate](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-database_url-with-hibernate)

If you are configuring Hibernate manually with a hibernate.cfg.xml outside of some other framework, you can set the JDBC URL in your Java code when building the service registry. For example:

Map<String,String> jdbcUrlSettings = new HashMap<>();

String jdbcDbUrl = System.getenv("JDBC\_DATABASE\_URL");

if (null != jdbcDbUrl) {

jdbcUrlSettings.put("hibernate.connection.url", System.getenv("JDBC\_DATABASE\_URL"));

}

registry = new StandardServiceRegistryBuilder().

configure("hibernate.cfg.xml").

applySettings(jdbcUrlSettings).

build();

This will override any URL configuring in the hibernate.cfg.xml.

## [Using SSL with PostgreSQL](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-ssl-with-postgresql)

We used to suggest adding the URL parameter sslmode=disable to JDBC URLs. We now [require use of SSL](https://devcenter.heroku.com/changelog-items/852) for all new Heroku Postgres databases. We will be enforcing use of SSL on all Heroku Postgres databases from March 2018. Please do not disable SSL for your database or your applications may break.

By default, Heroku will attempt to enable SSL for the PostgreSQL JDBC driver by setting the property sslmode=require globally. Use of SSL is required for all database connections on Heroku Postgres. The default sslmode is set in a small properties file that is automatically added to your classpath. You prevent this file from being injected by running heroku config:set SKIP\_PGCONFIG\_INSTALL=true.

Drivers for other database vendors are unaffected.

## [Connecting to a database remotely](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#connecting-to-a-database-remotely)

If you’re using a [Heroku Postgres](https://devcenter.heroku.com/categories/heroku-postgres) database you can connect to it remotely for maintenance and debugging purposes. However doing so requires that you use an SSL connection. Your JDBC connection URL will need to include the URL param:

sslmode=require

For example:

jdbc:postgresql://<host>:<port>/<dbname>?sslmode=require&user=<username>&password=<password>

If you do not add sslmode=require you will get a connection error.

See the [official PostgreSQL documentation](https://jdbc.postgresql.org/documentation/ssl/) for more information on SSL support with the Postgres JDBC driver. For other SQL databases, refer to your vendor specific JDBC documentation on how to configure SSL support.

It is important to add this parameter in code rather than editing the config var directly. Various automated events such as failover can change the config var, and edits there would be lost.

When connecting to a Private or Shield database via mTLS there are some additional parameters required in your JDBC connection URL. See our documentation on [connecting via mTLS](https://devcenter.heroku.com/articles/heroku-postgres-via-mtls#configuring-external-applications-to-connect-via-mtls) for further details.

## [Running database migrations](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#running-database-migrations)

Most databases will need to have their schema changed at some point. Migration tools such as Liquibase and Flyway control the version history of these changes. You can learn how to run these on Heroku in the article [Running Database Migrations for Java Apps](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps)

## [Using with Heroku Postgres Connection Pooling](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-with-heroku-postgres-connection-pooling)

When using JDBC with [Heroku Postgres Connection Pooling](https://devcenter.heroku.com/articles/postgres-connection-pooling), the JDBC\_DATABASE\_URL will be set to the value defined by the DATABASE\_CONNECTION\_POOL\_URL configuration variable.

## [Sample project](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#sample-project)

A sample project illustrating the various methods of setting up a database connection on Heroku can be found at: <https://github.com/heroku/devcenter-java-database>

To try it out first clone the git repository:

git clone git://github.com/heroku/devcenter-java-database.git

In the devcenter-java-database directory run the Maven build for the project:

mvn package

If you have a local Postgres database and want to test things locally, first set the DATABASE\_URL environment variable (using the correct values):

* On Linux/Mac:

export DATABASE\_URL=postgres://foo:foo@localhost/hellodb

* On Windows:

set DATABASE\_URL=postgres://foo:foo@localhost/hellodb

To run the example applications locally, execute the generated start scripts:

* On Linux/Mac:

sh devcenter-java-database-plain-jdbc/target/bin/main

sh devcenter-java-database-spring-xml/target/bin/main

sh devcenter-java-database-spring-java/target/bin/main

* On Windows:

devcenter-java-database-plain-jdbc/target/bin/main.bat

devcenter-java-database-spring-xml/target/bin/main.bat

devcenter-java-database-spring-java/target/bin/main.bat

For each command you should see a message like the following indicating that everything worked:

Read from DB: 2011-11-23 11:37:03.886016

To run on Heroku, first create a new application:

heroku create

Creating stark-sword-398... done, stack is heroku-18

http://stark-sword-398.herokuapp.com/ | git@heroku.com:stark-sword-398.git

Git remote heroku added

Then deploy the application on Heroku:

git push heroku master

Counting objects: 70, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (21/21), done.

Writing objects: 100% (70/70), 8.71 KiB, done.

Total 70 (delta 14), reused 70 (delta 14)

-----> Heroku receiving push

-----> Java app detected

-----> Installing Maven 3.0.3..... done

-----> executing /app/tmp/repo.git/.cache/.maven/bin/mvn -B -Duser.home=/tmp/build\_2y7ju7daa9t04 -Dmaven.repo.local=/app/tmp/repo.git/.cache/.m2/repository -DskipTests=true clean install

[INFO] Scanning for projects...

[INFO] ------------------------------------------------------------------------

[INFO] Reactor Build Order:

[INFO]

[INFO] devcenter-java-database-plain-jdbc

[INFO] devcenter-java-database-spring-xml

[INFO] devcenter-java-database-spring-java

[INFO] devcenter-java-database

[INFO]

[INFO] ------------------------------------------------------------------------

[INFO] Building devcenter-java-database-plain-jdbc 1.0-SNAPSHOT

[INFO] ------------------------------------------------------------------------

Downloading: http://s3pository.heroku.com/jvm/postgresql/postgresql/9.0-801.jdbc4/postgresql-9.0-801.jdbc4.pom

...

Now execute any of the examples on Heroku:

heroku run "sh devcenter-java-database-plain-jdbc/target/bin/main"

Running sh devcenter-java-database-plain-jdbc/target/bin/main attached to terminal... up, run.1

Read from DB: 2011-11-29 20:36:25.001468

## [Further learning](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#further-learning)

[Learn about the database options](https://devcenter.heroku.com/categories/heroku-postgres)

### [Keep reading](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#keep-reading)

* [Java Database Operations](https://devcenter.heroku.com/categories/java-database-operations)

### [Feedback](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fconnecting-to-relational-databases-on-heroku-with-java&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Database Connection Pooling with Java

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/database-connection-pooling-with-java)

Last updated August 27, 2020

## Table of Contents

* [Using Apache DBCP](https://devcenter.heroku.com/articles/database-connection-pooling-with-java#using-apache-dbcp)
* [Configuring the connection pool](https://devcenter.heroku.com/articles/database-connection-pooling-with-java#configuring-the-connection-pool)
* [Limit connections with PgBouncer](https://devcenter.heroku.com/articles/database-connection-pooling-with-java#limit-connections-with-pgbouncer)

Connection pooling is a pattern used by software applications to connect to databases using a pre-created set of reusable connection objects. When a new connection is required, an existing connection is retrieved from the pool. When the thread using the connection has completed, it is placed back in pool for use by another thread. This pattern reduces the overhead of connecting to a database by decreasing network traffic, limiting the cost of creating new connections, and reducing the load on the garbage collector.

Many Java Application Frameworks include their own connection pooling APIs. But the principles used to configure all frameworks are generally the same. In this article, you’ll learn how to create a database connection pool using the Java Database Connectivity (JDBC) API and the Apache DBCP pooling library.

If you already have a Java application, you may use it for this example. Otherwise, create a simple application from the [Getting Started with Java on Heroku](https://devcenter.heroku.com/articles/getting-started-with-java) article before proceeding. You should also be familiar with [Connecting to Relational Databases on Heroku with Java](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java).

## [Using Apache DBCP](https://devcenter.heroku.com/articles/database-connection-pooling-with-java#using-apache-dbcp)

Open your application’s pom.xml file and add the following libraries to the <dependencies> element. If you’re using the sample Getting Started application, then you might need to upgrade your postgresql dependency.

Apache DBCP 2 is only compatible with Java 7 and JDBC 4.1. If you’re using Java 6 or JDBC 4, then you’ll need to use DBCP 1.4

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<version>9.3-1102-jdbc41</version>

</dependency>

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-dbcp2</artifactId>

<version>2.0.1</version>

</dependency>

Now run Maven to download and install the new jar files:

mvn clean package

...

Downloading: http://repo.maven.apache.org/maven2/org/apache/commons/commons-dbcp2/2.0.1/commons-dbcp2-2.0.1.pom

Downloaded: http://repo.maven.apache.org/maven2/org/apache/commons/commons-dbcp2/2.0.1/commons-dbcp2-2.0.1.pom (14 KB at 16.7 KB/sec)

Downloading: http://repo.maven.apache.org/maven2/org/apache/commons/commons-parent/33/commons-parent-33.pom

Downloaded: http://repo.maven.apache.org/maven2/org/apache/commons/commons-parent/33/commons-parent-33.pom (52 KB at 254.1 KB/sec)

Downloading: http://repo.maven.apache.org/maven2/org/apache/commons/commons-pool2/2.2/commons-pool2-2.2.pom

Downloaded: http://repo.maven.apache.org/maven2/org/apache/commons/commons-pool2/2.2/commons-pool2-2.2.pom (12 KB at 110.2 KB/sec)

Downloading: http://repo.maven.apache.org/maven2/org/apache/commons/commons-dbcp2/2.0.1/commons-dbcp2-2.0.1.jar

Downloading: http://repo.maven.apache.org/maven2/org/apache/commons/commons-pool2/2.2/commons-pool2-2.2.jar

Downloaded: http://repo.maven.apache.org/maven2/org/apache/commons/commons-pool2/2.2/commons-pool2-2.2.jar (106 KB at 535.6 KB/sec)

Downloaded: http://repo.maven.apache.org/maven2/org/apache/commons/commons-dbcp2/2.0.1/commons-dbcp2-2.0.1.jar (165 KB at 474.1 KB/sec)

...

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 5.168 s

[INFO] Finished at: 2014-08-15T15:10:51-06:00

[INFO] Final Memory: 21M/230M

[INFO] ------------------------------------------------------------------------

Next, open the source code for the class that will create connections. If you’re working from the [Getting Started](https://github.com/heroku/java-getting-started) application, open the Main.java file. Add the following statements at the top of the file:

import java.sql.\*;

import org.apache.commons.dbcp.\*;

Then add a new instance variable to the Main class:

private BasicDataSource connectionPool;

This variable represents the connection pool. We will initialize and configure it when the Main object is instantiated. Let’s add that code to do that in the Main class’s constructor:

public Main() throws URISyntaxException, SQLException {

URI dbUri = new URI(System.getenv("DATABASE\_URL"));

String dbUrl = "jdbc:postgresql://" + dbUri.getHost() + dbUri.getPath();

connectionPool = new BasicDataSource();

if (dbUri.getUserInfo() != null) {

connectionPool.setUsername(dbUri.getUserInfo().split(":")[0]);

connectionPool.setPassword(dbUri.getUserInfo().split(":")[1]);

}

connectionPool.setDriverClassName("org.postgresql.Driver");

connectionPool.setUrl(dbUrl);

connectionPool.setInitialSize(1);

}

In this method, we’re retrieving the username, password and dbUrl as we would with a non-pooled connection. Then we’re initializing the connectionPool with those parameters and calling connectionPool.setInitialSize(3) to set the initial size of the pool. The BasicDataSource object will immediately create these connections for us and they will be ready to use when our application starts receiving traffic.

We can retrieve a connection from the pool like so:

Connection connection = connectionPool.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");

while (rs.next()) {

System.out.println("Read from DB: " + rs.getTimestamp("tick") + "\n");

}

Once we have a Connection, we use it exactly the same as any other JDBC Connection.

In addition to configuring the initial size of the connection pool, we might also want to set it’s maximum size, the maximum lifetime of the connections (before they is discarded and new ones replace them), or the maximum and minimum number of idle connections to keep before adjusting the size of pool. All of these can be set with [methods on the BasicDataSource class](https://commons.apache.org/proper/commons-dbcp/api-2.0.1/index.html).

You’ve learned how to configure the connection pool, but knowing what values to use when configure it is a different topic.

## [Configuring the connection pool](https://devcenter.heroku.com/articles/database-connection-pooling-with-java#configuring-the-connection-pool)

The number of idle connections to keep warm in your pool depends on the size and nature of your application. Many users find one connection per thread handling HTTP requests is sufficient (assuming threads handling HTTP requests are the only threads using connections). Your application may need more if it experiences very high throughput such that it can’t turn connections over to new threads quick enough. Or you may need fewer if not every HTTP request needs to access the database. Ultimately, profiling your application under production loads is the best way to determine the appropriate pool parameters.

In development you can see the number of connections used by your application by checking the database.

psql -h localhost

psql (9.3.2)

Type "help" for help.

jkutner=# \q

This will open a connection to your development database. You can then see the number of connections to your postgres 9.1 or previous database by running:

select count(\*) from pg\_stat\_activity where procpid <> pg\_backend\_pid() and usename = current\_user;

On Postgres 9.2 and later the command is:

select count(\*) from pg\_stat\_activity where pid <> pg\_backend\_pid() and usename = current\_user;

Which will return with the number of connections on that database:

count

-------

5

(1 row)

Under simulated production loads, this will give you a good indication of what size pool you need. There are, however, some constraints.

### [Maximum database connections](https://devcenter.heroku.com/articles/database-connection-pooling-with-java#maximum-database-connections)

Heroku provides [managed Postgres](https://elements.heroku.com/addons/heroku-postgresql) databases. Different tiered databases have different connection limits, which you can find listed on the [Heroku Postgres add-on documentation](https://elements.heroku.com/addons/heroku-postgresql). Databases in the lower tiers permit fewer connections than databases in the higher tiers. Once your database has the maximum number of active connections, it will no longer accept new connections. This will result in connection timeouts from your application and will likely cause exceptions.

When scaling out, it is important to keep in mind how many active connections your application needs. If each dyno allows 5 database connections, you can only scale out to four dynos before you need to provision a more robust database.

Now that you know how to configure your connection pool and how to figure out how many connections your database can handle you will need to calculate the right number of connections that each dyno will need.

## [Limit connections with PgBouncer](https://devcenter.heroku.com/articles/database-connection-pooling-with-java#limit-connections-with-pgbouncer)

You can continue to scale out your applications with additional dynos until you have reached your database connection limits. Before you reach this point it is recommended to limit the number of connections required by each dyno by using the [PgBouncer buildpack](https://github.com/heroku/heroku-buildpack-pgbouncer).

PgBouncer maintains a pool of connections that your database transactions share. This keeps connections to Postgres, which are otherwise open and idle, to a minimum. However, transaction pooling prevents you from using named prepared statements, session advisory locks, listen/notify, or other features that operate on a session level. See the [PgBouncer buildpack FAQ for full list of limitations](https://github.com/heroku/heroku-buildpack-pgbouncer#faq) for more information.

For many frameworks, you must [disable prepared statements](https://github.com/heroku/heroku-buildpack-pgbouncer#disable-prepared-statements) in order to use PgBouncer. Then set your app to use the PgBouncer buildpack.

For JDBC, this requires adding prepareThreshold=0 to the connection string. But it may also be necessary to [patch your JDBC driver](https://pgbouncer.github.io/faq.html#how-to-use-prepared-statements-with-transaction-pooling).

Do not continue before disabling prepared statements, or verifying that your framework is not using them.

heroku buildpacks:add heroku/pgbouncer

Next we need to ensure your application can run so you need to add your language specific buildpack. Since you are using Java it would be:

heroku buildpacks:add heroku/java

Now you must modify your Procfile to start PgBouncer. In your Procfile add the command bin/start-pgbouncer-stunnel to the beginning of your web entry. So if your Procfile was

web: java $JAVA\_OPTS -cp target/classes:target/dependency/\* Main

Will now be:

web: bin/start-pgbouncer-stunnel java $JAVA\_OPTS -cp target/classes:target/dependency/\* Main

Commit the results to git, test on a staging app, and then deploy to production.

When deploying you should see this in the output:

=====> Detected Framework: pgbouncer-stunnel

For more information on connection pooling with Java, JDBC and Apache DBCP, see the [Apache Commons Website](https://commons.apache.org/proper/commons-dbcp/).

You can find the [source code for the examples used in this article](https://github.com/heroku/devcenter-java-database) on GitHub.

### [Keep reading](https://devcenter.heroku.com/articles/database-connection-pooling-with-java#keep-reading)

* [Java Database Operations](https://devcenter.heroku.com/categories/java-database-operations)

### [Feedback](https://devcenter.heroku.com/articles/database-connection-pooling-with-java#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fdatabase-connection-pooling-with-java&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Running Database Migrations for Java Apps

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/running-database-migrations-for-java-apps)

Last updated June 05, 2020

## Table of Contents

* [Using Liquibase](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps#using-liquibase)
* [Using Flyway](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps#using-flyway)
* [Further Reading](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps#further-reading)

Most database-backed applications will need to change their database schema during the course of operations. These changes are often controlled by a process called migrations or evolutions.

In this article, you’ll learn how to use two tools, [Liquibase](https://www.liquibase.org) and [Flyway](https://flywaydb.org/), to run database migrations for a Java application on Heroku.

## [Using Liquibase](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps#using-liquibase)

There are many ways to run Liquibase. It provides a Maven plugin, a standalone command-line tool, a Hibernate plugin, and a Spring Bean. In this article, we’ll discuss the best mechanisms for use on Heroku.

### [Running Liquibase automatically with Spring](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps#running-liquibase-automatically-with-spring)

Liquibase provides a very convenient [SpringLiquibase bean](https://docs.liquibase.com/tools-integrations/springboot/using-springboot-with-maven.html) that automatically runs your migrations on startup if they are in the correct location.

If you are using Spring Boot, you only need to include the liquibase-core dependency and put your change log in src/resources/db/changelog/db.changelog-master.yaml, as demonstrated in this [Spring and Liquibase sample application](https://github.com/kissaten/liquibase-example).

Upon startup, you will see something like this in your logs:

2015-10-20T02:00:51.937179+00:00 app[web.1]: 2015-10-20 02:00:51.936 INFO 3 --- [main] liquibase : Successfully acquired change log lock

2015-10-20T02:00:55.419669+00:00 app[web.1]: 2015-10-20 02:00:55.419 INFO 3 --- [main] liquibase : Reading from public.databasechangelog

2015-10-20T02:00:55.571104+00:00 app[web.1]: 2015-10-20 02:00:55.555 INFO 3 --- [main] liquibase : Successfully released change log lock

However, running migrations at startup does add to your app’s boot time and may cause you to exceed the boot-timeout limit imposed by Heroku. If so, you may find that running the migrations in the Heroku release phase is preferable.

### [Running Liquibase with the Maven Plugin](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps#running-liquibase-with-the-maven-plugin)

The Liquibase migrations can be run in [Heroku release phase](https://devcenter.heroku.com/articles/release-phase) using the Liquibase Maven Plugin by adding the following plugin configuration to your pom.xml:

<plugin>

<groupId>org.liquibase</groupId>

<artifactId>liquibase-maven-plugin</artifactId>

<configuration>

<changeLogFile>src/main/resources/db/changelog/db.changelog-master.yaml</changeLogFile>

<url>${env.JDBC\_DATABASE\_URL}</url>

</configuration>

</plugin>

Next, make sure you’re using the [Maven Wrapper](https://github.com/takari/maven-wrapper) in your project. If you’re not already, you can add it by running:

mvn -N io.takari:maven:wrapper

git add mvnw .mvn

git commit -m "Added maven wrapper"

With this in place, you can add a process entry to your Procfile that will run the migrations in the release phase of deployment:

release: ./mvnw liquibase:update

However, this will require downloading all of the Liquibase dependencies each time release phase runs because the Maven .m2 cache is not included in your app’s [slug](https://devcenter.heroku.com/articles/slug-compiler). If that overhead is prohibitive you may want to run Liquibase without Maven.

### [Running Liquibase with the command-line tool](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps#running-liquibase-with-the-command-line-tool)

Your Liquibase migrations can be run in [Heroku release phase](https://devcenter.heroku.com/articles/release-phase) using the Liquibase command-line tool. To use this tool, you’ll need to include the Liquibase JAR file in your slug by adding the following plugin configuration to your Maven pom.xml:

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-dependency-plugin</artifactId>

<executions>

<execution>

<id>copy-dependencies</id>

<phase>package</phase>

<goals><goal>copy</goal></goals>

<configuration>

<artifactItems>

<artifactItem>

<groupId>org.liquibase</groupId>

<artifactId>liquibase-core</artifactId>

<version>3.4.1</version>

<destFileName>liquibase.jar</destFileName>

</artifactItem>

<artifactItem>

<groupId>org.yaml</groupId>

<artifactId>snakeyaml</artifactId>

<version>1.13</version>

<outputDirectory>${project.build.directory}/dependency/lib</outputDirectory>

</artifactItem>

<artifactItem>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<version>9.4-1204-jdbc41</version>

<destFileName>postgres.jar</destFileName>

</artifactItem>

</artifactItems>

</configuration>

</execution>

</executions>

</plugin>

This will copy the JAR file to target/dependency/liquibase.jar during the Maven package goal. With this in place, you can add a process entry to your Procfile that will run the migrations in the release phase of deployment:

release: java -jar target/dependency/liquibase.jar --changeLogFile=src/main/resources/db/changelog/db.changelog-master.yaml --url=$JDBC\_DATABASE\_URL --classpath=target/dependency/postgres.jar update

Liquibase is not the only migration engine for Java. You can also use Flyway.

## [Using Flyway](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps#using-flyway)

Flyway provides several mechanisms for running migrations including a command-line tool, Maven plugin, Gradle plugin and an SBT plugin. In this article, we’ll discuss the best mechanisms for use on Heroku.

### [Running Flyway automatically with Spring](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps#running-flyway-automatically-with-spring)

You can run Flyway automatically by including the flyway-core dependency in your app and putting your migration scripts in src/main/resources/db/migration/, as demonstrated in this [Spring and Flyway sample application](https://github.com/kissaten/flyway-example).

However, running migrations at startup does add to your app’s boot time and may cause you to exceed the boot-timeout limit imposed by Heroku. If this is the case, you may find that running the migrations from Heroku release phase preferable.

### [Running Flyway with the Maven Plugin](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps#running-flyway-with-the-maven-plugin)

You can also run the Flyway Maven Plugin from Heroku’s release phase. First, add the plugin to you pom.xml:

<plugin>

<groupId>org.flywaydb</groupId>

<artifactId>flyway-maven-plugin</artifactId>

<configuration>

<url>${env.JDBC\_DATABASE\_URL}</url>

</configuration>

</plugin>

Next, make sure you’re using the [Maven Wrapper](https://github.com/takari/maven-wrapper) in your project. If you’re not already, you can add it by running:

mvn -N io.takari:maven:wrapper

git add mvnw .mvn

git commit -m "Added maven wrapper"

With this in place, you can add a process entry to your Procfile that will run the migrations in the release phase of deployment:

release: ./mvnw flyway:migrate

However, this will require downloading all of the Flyway dependencies each time release phase runs because the Maven .m2 cache is not included in your app’s [slug](https://devcenter.heroku.com/articles/slug-compiler). If that overhead is prohibitive you may want to run Flyway without Maven.

### [Running Flyway with the Java API](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps#running-flyway-with-the-java-api)

To use the Flyway Java API, create a simple class with a main method, such as this:

package sample.flyway;

import org.flywaydb.core.Flyway;

public class Migrations {

public static void main(String[] args) throws Exception {

Flyway flyway = new Flyway();

flyway.setDataSource(System.getenv("JDBC\_DATABASE\_URL"),

System.getenv("JDBC\_DATABASE\_USERNAME"),

System.getenv("JDBC\_DATABASE\_PASSWORD"));

flyway.migrate();

}

}

Then copy the flyway JAR file into your slug file by adding this plugin configuration to your pom.xml file:

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-dependency-plugin</artifactId>

<executions>

<execution>

<id>copy-dependencies</id>

<phase>package</phase>

<goals><goal>copy</goal></goals>

<configuration>

<artifactItems>

<artifactItem>

<groupId>org.flywaydb</groupId>

<artifactId>flyway-core</artifactId>

<version>3.2.1</version>

<destFileName>flyway.jar</destFileName>

</artifactItem>

<artifactItem>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<version>9.4-1204-jdbc41</version>

<destFileName>postgres.jar</destFileName>

</artifactItem>

</artifactItems>

</configuration>

</execution>

</executions>

</plugin>

This will copy the JAR file to target/dependency/flyway.jar during the Maven package goal. With this in place, you can add a process entry to your Procfile that will run the migrations:

release: java -cp target/spring-boot-sample-flyway-1.0.0.jar:target/dependency/\* sample.flyway.Migrations

This will run the migrations during the [Heroku release phase](https://devcenter.heroku.com/articles/release-phase). You can also run the migrations manually with this command:

heroku run release

## [Further Reading](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps#further-reading)

For more information, see the official documentation pages for each of these open source projects:

* [Liquibase](https://www.liquibase.org/get-started)
* [Flyway](https://flywaydb.org/documentation/)

And you can read more about [Connecting to Relational Databases on Heroku with Java](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java) in the Dev Center.

### [Keep reading](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps#keep-reading)

* [Java Database Operations](https://devcenter.heroku.com/categories/java-database-operations)

### [Feedback](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Frunning-database-migrations-for-java-apps&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Create a Java Web Application Using Embedded Tomcat

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/create-a-java-web-application-using-embedded-tomcat)

Last updated December 16, 2019

## Table of Contents

* [Prerequisites](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#prerequisites)
* [Create your pom.xml](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#create-your-pom-xml)
* [Add a launcher class](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#add-a-launcher-class)
* [Add a Servlet](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#add-a-servlet)
* [Add a JSP](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#add-a-jsp)
* [Run your application](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#run-your-application)
* [Deploy your application to Heroku](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#deploy-your-application-to-heroku)
* [Create a Procfile](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#create-a-procfile)
* [Deploy to Heroku](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#deploy-to-heroku)
* [Clone the source](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#clone-the-source)

This tutorial will show you how to create a simple Java web application using embedded Tomcat.

Follow each step to build an app from scratch, or skip to the end get the source for this article.

Sample code for the [embedded Tomcat demo](https://github.com/heroku/devcenter-embedded-tomcat) is available on GitHub.

## [Prerequisites](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#prerequisites)

* Basic Java knowledge, including an installed version of the JVM and Maven.
* Basic Git knowledge, including an installed version of Git.

## [Create your pom.xml](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#create-your-pom-xml)

Create a folder to hold your app and create a file called pom.xml in the root of that folder with the following contents:

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.heroku.sample</groupId>

<artifactId>embeddedTomcatSample</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<name>embeddedTomcatSample Maven Webapp</name>

<url>http://maven.apache.org</url>

<properties>

<tomcat.version>8.5.23</tomcat.version>

</properties>

<dependencies>

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-core</artifactId>

<version>${tomcat.version}</version>

</dependency>

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

<version>${tomcat.version}</version>

</dependency>

<dependency>

<groupId>org.apache.tomcat</groupId>

<artifactId>tomcat-jasper</artifactId>

<version>${tomcat.version}</version>

</dependency>

<dependency>

<groupId>org.apache.tomcat</groupId>

<artifactId>tomcat-jasper-el</artifactId>

<version>${tomcat.version}</version>

</dependency>

<dependency>

<groupId>org.apache.tomcat</groupId>

<artifactId>tomcat-jsp-api</artifactId>

<version>${tomcat.version}</version>

</dependency>

</dependencies>

<build>

<finalName>embeddedTomcatSample</finalName>

<plugins>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>appassembler-maven-plugin</artifactId>

<version>2.0.0</version>

<configuration>

<assembleDirectory>target</assembleDirectory>

<programs>

<program>

<mainClass>launch.Main</mainClass>

<name>webapp</name>

</program>

</programs>

</configuration>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>assemble</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

This pom.xml defines the dependencies that you’ll need to run Tomcat in an embedded mode.

The last 3 entries are only required for applications that use JSP files. If you use this technique for an application that doesn’t use JSPs then you can just include the first 3 dependencies.

There is also a single plugin defined. The appassembler plugin generates a launch script that automatically sets up your classpath and calls your main method (created below) to launch your application.

## [Add a launcher class](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#add-a-launcher-class)

Create a file called Main.java in your src/main/java/launch directory and put the following in it:

package launch;

import java.io.File;

import org.apache.catalina.WebResourceRoot;

import org.apache.catalina.core.StandardContext;

import org.apache.catalina.startup.Tomcat;

import org.apache.catalina.webresources.DirResourceSet;

import org.apache.catalina.webresources.StandardRoot;

public class Main {

public static void main(String[] args) throws Exception {

String webappDirLocation = "src/main/webapp/";

Tomcat tomcat = new Tomcat();

//The port that we should run on can be set into an environment variable

//Look for that variable and default to 8080 if it isn't there.

String webPort = System.getenv("PORT");

if(webPort == null || webPort.isEmpty()) {

webPort = "8080";

}

tomcat.setPort(Integer.valueOf(webPort));

StandardContext ctx = (StandardContext) tomcat.addWebapp("/", new File(webappDirLocation).getAbsolutePath());

System.out.println("configuring app with basedir: " + new File("./" + webappDirLocation).getAbsolutePath());

// Declare an alternative location for your "WEB-INF/classes" dir

// Servlet 3.0 annotation will work

File additionWebInfClasses = new File("target/classes");

WebResourceRoot resources = new StandardRoot(ctx);

resources.addPreResources(new DirResourceSet(resources, "/WEB-INF/classes",

additionWebInfClasses.getAbsolutePath(), "/"));

ctx.setResources(resources);

tomcat.start();

tomcat.getServer().await();

}

}

This does just what is enough to launch the server. The sample application contains a more complete version of code that handles temp directories and other things.

## [Add a Servlet](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#add-a-servlet)

Create a file called HelloServlet.java in the src/main/java/servlet directory and put the following into it:

package servlet;

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.ServletOutputStream;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet(

name = "MyServlet",

urlPatterns = {"/hello"}

)

public class HelloServlet extends HttpServlet {

@Override

protected void doGet(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, IOException {

ServletOutputStream out = resp.getOutputStream();

out.write("hello heroku".getBytes());

out.flush();

out.close();

}

}

This is simple Servlet that uses annotations to configure itself.

## [Add a JSP](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#add-a-jsp)

Create a file called index.jsp in the src/main/webapp directory:

<html>

<body>

<h2>Hello Heroku!</h2>

</body>

</html>

## [Run your application](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#run-your-application)

To generate the start scripts simply run:

mvn package

And then simply run the script. On Mac and Linux, the command is:

sh target/bin/webapp

On Windows the command is:

C:/> target/bin/webapp.bat

That’s it. Your application should start up on port 8080. You can see the JSP at http://localhost:8080 and the servlet and http://localhost:8080/hello

## [Deploy your application to Heroku](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#deploy-your-application-to-heroku)

## [Create a Procfile](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#create-a-procfile)

You declare how you want your application executed in Procfile in the project root. Create this file with a single line:

web: sh target/bin/webapp

Learn more about [procfile](https://devcenter.heroku.com/articles/procfile).

## [Deploy to Heroku](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#deploy-to-heroku)

You can either deploy to Heroku by using the [Heroku Maven plugin](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin) or you can deploy using Git. The latter is described in this article.

Commit your changes to Git:

git init

git add .

git commit -m "Ready to deploy"

Create the app:

heroku create

Creating high-lightning-129... done, stack is heroku-18

http://high-lightning-129.herokuapp.com/ | git@heroku.com:high-lightning-129.git

Git remote heroku added

Deploy your code:

git push heroku master

Counting objects: 227, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (117/117), done.

Writing objects: 100% (227/227), 101.06 KiB, done.

Total 227 (delta 99), reused 220 (delta 98)

-----> Heroku receiving push

-----> Java app detected

...

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 36.612s

[INFO] Finished at: Tue Aug 30 04:03:02 UTC 2011

[INFO] Final Memory: 19M/287M

[INFO] ------------------------------------------------------------------------

-----> Discovering process types

Procfile declares types -> web

-----> Compiled slug size is 62.7MB

-----> Launching... done, v5

http://pure-window-800.herokuapp.com deployed to Heroku

Congratulations! Your web app should now be up and running on Heroku. Open it in your browser with:

heroku open

This will show your your JSP and then you can navigate to /hello to see your servlet.

## [Clone the source](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#clone-the-source)

If you want to skip the creation steps you can clone the finished sample:

git clone https://github.com/heroku/devcenter-embedded-tomcat

### [Keep reading](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#keep-reading)

* [Java Advanced Topics](https://devcenter.heroku.com/categories/java-advanced-topics)

### [Feedback](https://devcenter.heroku.com/articles/create-a-java-web-application-using-embedded-tomcat#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fcreate-a-java-web-application-using-embedded-tomcat&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Warming Up a Java Process

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/warming-up-a-java-process)

Last updated December 16, 2019

## Table of Contents

* [Prebooting an application](https://devcenter.heroku.com/articles/warming-up-a-java-process#prebooting-an-application)
* [Creating a warm-up script](https://devcenter.heroku.com/articles/warming-up-a-java-process#creating-a-warm-up-script)
* [Enabling the warm-up script](https://devcenter.heroku.com/articles/warming-up-a-java-process#enabling-the-warm-up-script)
* [Customizing the warm-up script](https://devcenter.heroku.com/articles/warming-up-a-java-process#customizing-the-warm-up-script)
* [Further reading](https://devcenter.heroku.com/articles/warming-up-a-java-process#further-reading)

The first request made to a Java or JVM web application is often substantially slower than the average response time over the life of the process. This warm-up period can usually be attributed to lazy class loading and just-in-time compilation, which optimize the JVM for subsequent requests that execute identical code.

As a result, many users find it beneficial to submit a few artificial requests into the JVM process before using it to handle real traffic. Even a simple set of no-op requests to your application can warm-up the networking and messaging parts of the stack, which usually constitute a large portion of the request overhead.

This article describes one strategy for implementing such behavior on Heroku.

## [Prebooting an application](https://devcenter.heroku.com/articles/warming-up-a-java-process#prebooting-an-application)

In order to do anything with a process before the router starts sending it requests, you must enable [Heroku’s Preboot feature](https://devcenter.heroku.com/articles/preboot). This will provide some time between when the app starts up and when it begins handling real traffic.

## [Creating a warm-up script](https://devcenter.heroku.com/articles/warming-up-a-java-process#creating-a-warm-up-script)

With pre-boot enabled, you can create a script that will run alongside your application and send some requests before it’s brought into the pool of active dynos.

Add a bin/ directory in the root of your project, and create a bin/warmup file. Open that file in your favorite editor and add the following code:

until $(curl -o /dev/null -s -I -f http://localhost:$PORT); do

sleep 5

done

This ensures that the script waits until the main process has started up and bound to $PORT before sending any requests.

To send the requests, add this code next:

for ROUTE in $WARMUP\_ROUTES; do

echo "[warmup] calling $ROUTE"

curl -L "http://localhost:$PORT$ROUTE" >/dev/null 2>&1

done

This sends a single request to the routes defined by the $WARMUP\_ROUTES config variable, which you can set like this:

heroku config:set WARMUP\_ROUTES="/ /hello /db"

This will send warm-up requests to three different routes: /, /hello and /db. You will need to customize this variable to suit your application’s routes. Some users prefer to add special routes specifically for warm-up. It is not required to execute application specific logic, but it can help.

After you’ve chosen the routes, you need to add this script to your Procfile.

## [Enabling the warm-up script](https://devcenter.heroku.com/articles/warming-up-a-java-process#enabling-the-warm-up-script)

To use the bin/warmup script, modify your existing Procfile command for the web process by adding the sh bin/warmup & prefix to it. For example, you might have:

web: sh bin/warmup & java -jar my-app.jar

This will start the warm-up script in the background, and the java process in the foreground.

Note that if you are running Windows locally, this script won’t work when you run heroku local because it’s specific to Linux platforms (like the one on Heroku). In this case, you’ll need to create a Procfile.windows and add it to the command by running heroku local -f Procfile.windows. The Windows Procfile will contain only the java command for the web: entry.

Once this is complete, add your changes to Git and redeploy:

git add bin/warmup Procfile

git commit -m "warmup"

git push heroku master

When your app restarts, it will behave as normal, but you’ll see a few [warmup] statements in the logs, which correspond to the routes you defined earlier.

You can find a complete example of an application that is set up this way in the [sample app on Github](https://github.com/kissaten/warmup-jvm).

## [Customizing the warm-up script](https://devcenter.heroku.com/articles/warming-up-a-java-process#customizing-the-warm-up-script)

Because the bin/warmup script is written in [Bash](https://en.wikipedia.org/wiki/Bash_%28Unix_shell%29), it can be customized by writing a few lines of code.

For example, if you want to repeat the warm-up requests multiple times, you could add for loop around the curl command. For example:

for i in {1..10}; do

curl ...

done

It is reasonable to repeat each warm-up request as much as a few hundred times, though not usually necessary.

If you’d like to make a more robust warm-up script, you can use a language other than Bash. Ruby and Python are both available on the [Cedar stack](https://devcenter.heroku.com/articles/stack), and you can even write a simple JVM app for this purpose.

## [Further reading](https://devcenter.heroku.com/articles/warming-up-a-java-process#further-reading)

Advanced users may want to configure the JVM so that just-in-time compilation is better optimized for their application and thus improves warm-up time. A great resource for this is [Java Performance: The Definitive Guide](https://www.safaribooksonline.com/library/view/java-performance-the/9781449363512/) by Scott Oaks.

Please see the Dev Center for more information on [Heroku’s Java Support](https://devcenter.heroku.com/articles/java-support).

### [Keep reading](https://devcenter.heroku.com/articles/warming-up-a-java-process#keep-reading)

* [Java Advanced Topics](https://devcenter.heroku.com/categories/java-advanced-topics)

### [Feedback](https://devcenter.heroku.com/articles/warming-up-a-java-process#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fwarming-up-a-java-process&utm_campaign=login&utm_medium=feedback&utm_source=web)

# JVM Runtime Metrics

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/language-runtime-metrics-jvm)

Last updated November 28, 2022

## Table of Contents

* [General Information](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#general-information)
* [Getting Started](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#getting-started)
* [Available Metrics](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#available-metrics)
* [Performance implications](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#performance-implications)
* [Using with JRuby](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#using-with-jruby)
* [Disabling Metrics Collection](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#disabling-metrics-collection)

The JVM Runtime Metrics feature allows you to view heap and non-heap memory, as well as garbage collection activity, for any app that runs inside of the Java Virtual Machine (JVM). This includes (but is not limited to) apps written in Java, Clojure, Gradle, Lein, Maven, Scala, and JRuby,

JVM Runtime Metrics are available for all dynos except for eco dynos.

## [General Information](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#general-information)

For general information on language specific metrics, please refer to the language runtime metrics [parent document](https://devcenter.heroku.com/articles/language-runtime-metrics).

## [Getting Started](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#getting-started)

Once you have enabled the Enhanced Language Metrics feature (by following [these instructions](https://devcenter.heroku.com/articles/language-runtime-metrics#setup-instructions)), re-deploy your application using an empty commit.

git commit --allow-empty -m "Enable Heroku Metrics"

git push heroku master

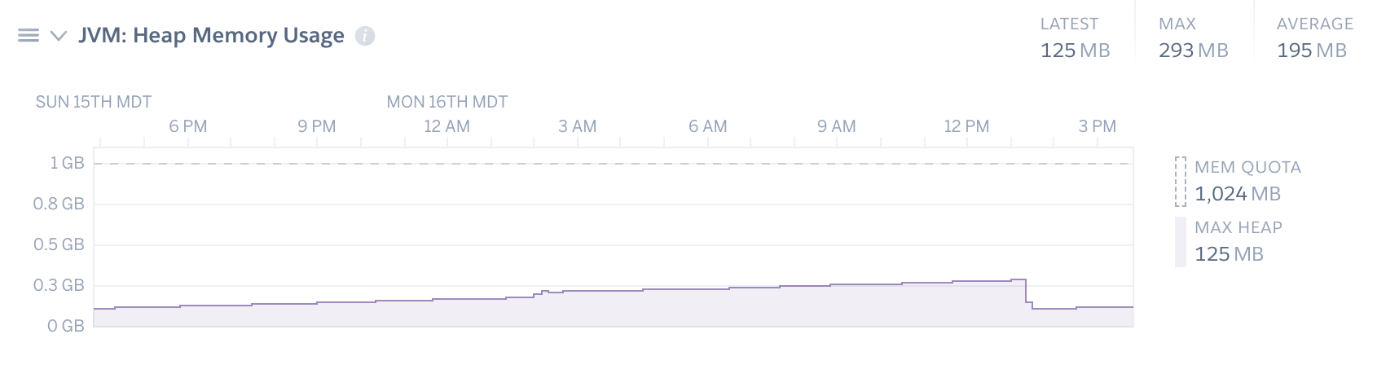
Afterwards, your application should begin exporting memory metrics for the JVM.

## [Available Metrics](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#available-metrics)

It may take a few minutes for these metrics to become available after the steps above are completed.

### [Heap Memory Usage](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#heap-memory-usage)

This chart plots, in megabytes (MB) the overall [memory quota](https://devcenter.heroku.com/articles/dynos#memory-behavior) and the maximum heap memory usage across all dynos in the currently selected process type. Summary metrics for the selected time interval, including latest, average and maximum memory, are also displayed.

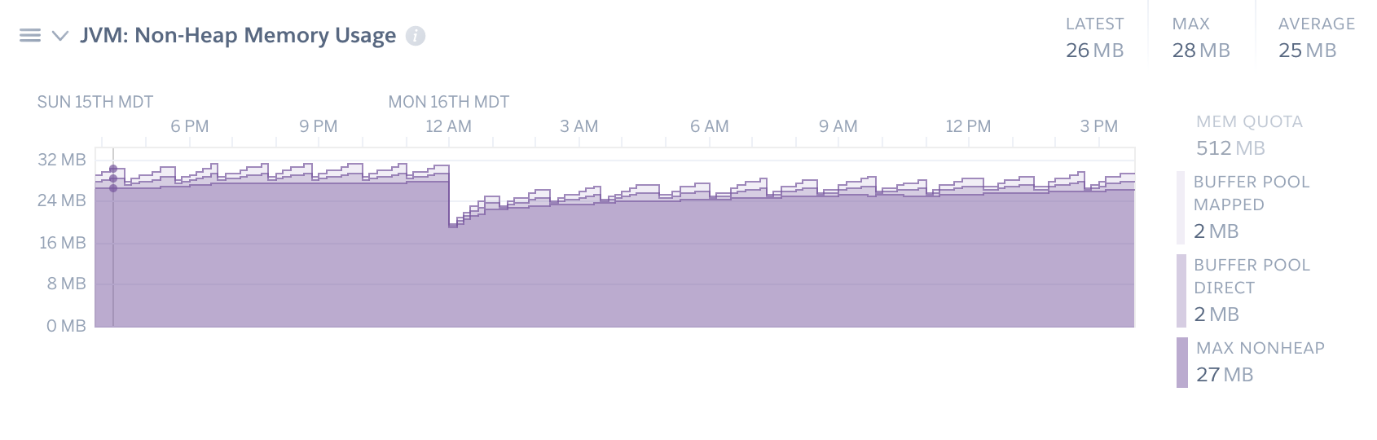


#### [How to use this chart](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#heap-memory-usage-how-to-use-this-chart)

Heap memory is where dynamically allocated memory resides. If you see heap memory growing unexpectedly, that’s typically an indication of a memory leak. You can use tools like [Heroku Java CLI](https://devcenter.heroku.com/articles/exec#using-java-debugging-tools) for further diagnosis.

### [Non-Heap Memory Usage](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#non-heap-memory-usage)

This chart plots, in megabytes (MB) the overall [memory quota](https://devcenter.heroku.com/articles/dynos#memory-behavior) (note: the memory quota time series is toggled off in this screenshot), the maximum non-heap memory usage, the maximum buffer pool direct memory usage, and the maximum buffer pool mapped memory usage, across all dynos in the currently selected process type. Summary metrics for the selected time interval, including latest, average and maximum memory, are also displayed.



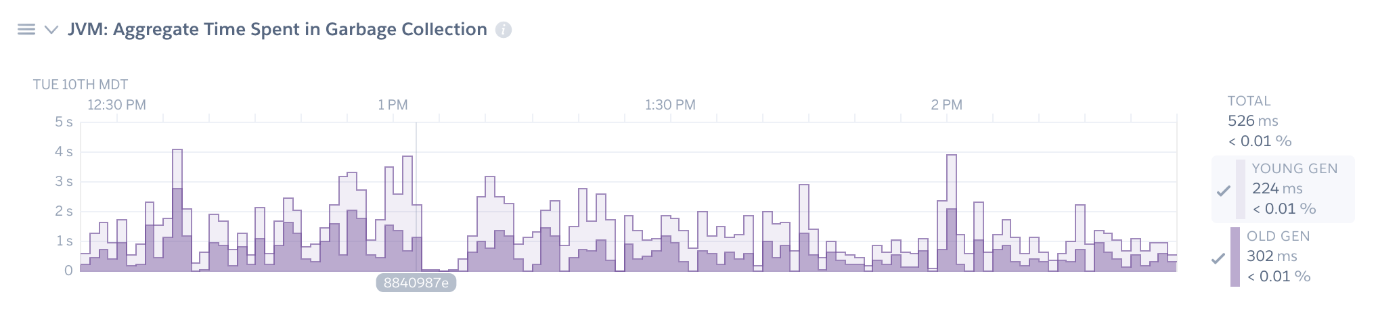
The buffer pool mapped and direct time series will only show up if you have any memory usage of those types during the selected timeframe.

#### [How to Use This Chart](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#non-heap-memory-usage-how-to-use-this-chart)

The non-heap memory usage chart can be used to identify memory leaks or excessive consumption of Metaspace and buffer pools. For more information on debugging these kinds of problems, see the article on [Troubleshooting Memory Issues in Java Applications](https://devcenter.heroku.com/articles/java-memory-issues).

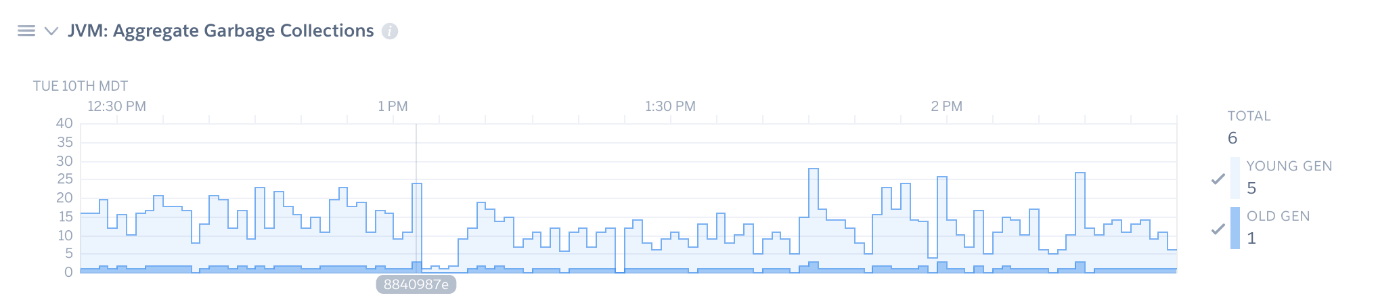
### [Aggregate Time Spent in Garbage Collection](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#aggregate-time-spent-in-garbage-collection)

This chart plots the aggregate amount of time that was spent doing garbage collection during each [roll up](https://devcenter.heroku.com/articles/metrics#data-resolution), broken down by Young and Old generations. In the legend, you will see a breakdown of the percentage of time during the roll up was spent running the garbage collector.



### [Aggregate Garbage Collections](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#aggregate-garbage-collections)

This chart plots the aggregate number of times that the garbage collector ran during each [roll up](https://devcenter.heroku.com/articles/metrics#data-resolution), broken down by Young and Old generations.



#### [How to Use These Charts](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#how-to-use-these-charts)

The garbage collection charts can be used to tune both the GC and the heap. Like many performance issues, choosing a heap size is a matter of balance. If the heap is too small, the program will spend too much time performance GC. But a very large heap will increase the duration of those pauses. A good rule of thumb is to size the heap so that it is 30% occupied after a full GC, as indicated by the GC chart.

## [Performance implications](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#performance-implications)

Language runtime metrics collects data from the JVM running on your dyno. The overall impact of JVM metrics collection on application performance is minimal. In our testing we observed the following:

* an estimated 1MB of additional peak heap usage
* ~200 additional classes loaded, resulting in < 1MB of additional Metaspace (off-heap) memory
* 1 additional thread

## [Using with JRuby](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#using-with-jruby)

To use this feature with JRuby, you must include the heroku/jvm buildpack with your app by running:

heroku buildpacks:add -i 1 heroku/jvm

Then redeploy your application.

## [Disabling Metrics Collection](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#disabling-metrics-collection)

To disable JVM metrics collection, simply toggle off the Enhanced Language Metrics toggle via the Metrics Preferences panel, or using this CLI command:

heroku labs:disable "runtime-heroku-metrics" -a "my-app-name"

### [Keep reading](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#keep-reading)

* [Java Advanced Topics](https://devcenter.heroku.com/categories/java-advanced-topics)

### [Feedback](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Flanguage-runtime-metrics-jvm&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Customizing the JDK

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/customizing-the-jdk)

Last updated December 15, 2022

## Table of Contents

* [Prerequisites](https://devcenter.heroku.com/articles/customizing-the-jdk#prerequisites)
* [Specify a JDK version](https://devcenter.heroku.com/articles/customizing-the-jdk#specify-a-jdk-version)
* [Create a .jdk-overlay folder](https://devcenter.heroku.com/articles/customizing-the-jdk#create-a-jdk-overlay-folder)
* [Add custom files](https://devcenter.heroku.com/articles/customizing-the-jdk#add-custom-files)
* [Adding custom certificates](https://devcenter.heroku.com/articles/customizing-the-jdk#adding-custom-certificates)
* [Verify the copy](https://devcenter.heroku.com/articles/customizing-the-jdk#verify-the-copy)
* [Other examples](https://devcenter.heroku.com/articles/customizing-the-jdk#other-examples)

There are some cases where files need to be bundled with the JDK in order to expose functionality in the runtime JVM. For example, the inclusion of a custom certificate authority (CA) store is common. To handle such cases, Heroku will copy files designated by the app in a .jdk-overlay folder into the JDK’s directory structure.

To include additional files in the JVM, follow these instructions:

## [Prerequisites](https://devcenter.heroku.com/articles/customizing-the-jdk#prerequisites)

* [A Java app running on Heroku.](https://devcenter.heroku.com/articles/getting-started-with-java)

## [Specify a JDK version](https://devcenter.heroku.com/articles/customizing-the-jdk#specify-a-jdk-version)

Create a system.properties file if one does not already exist, specify the version, and commit it to git. Supported version are described in the [Java Support article](https://devcenter.heroku.com/articles/java-support#supported-java-versions). The file’s contents should look something like this:

java.runtime.version=11

Then add the file to Git by running:

git add system.properties

git commit -m "JDK 11"

## [Create a .jdk-overlay folder](https://devcenter.heroku.com/articles/customizing-the-jdk#create-a-jdk-overlay-folder)

In your application’s root directory, create a .jdk-overlay folder.

mkdir .jdk-overlay

ls -la

total 24

drwxr-xr-x 9 user staff 306 Oct 16 14:43 .

drwxr-xr-x 202 user staff 6868 Oct 16 14:40 ..

drwxr-xr-x 13 user staff 442 Oct 16 15:06 .git

drwxr-xr-x 3 user staff 102 Oct 16 14:43 .jdk-overlay

-rw-r--r-- 1 user staff 45 Oct 16 14:40 Procfile

-rw-r--r-- 1 user staff 1860 Oct 16 14:40 pom.xml

drwxr-xr-x 3 user staff 102 Oct 16 14:40 src

-rw-r--r-- 1 user staff 25 Oct 16 14:40 system.properties

## [Add custom files](https://devcenter.heroku.com/articles/customizing-the-jdk#add-custom-files)

Copy any custom files into the .jdk-overlay directory. The files will be copied to their equivalent directory in the JDK. For example, to define a custom security policy, the java.policy file could be placed in the .jdk-overlay/jre/lib/security/ (.jdk-overlay/lib/security for Java 9 and higher) directory of your app’s repository.

## [Adding custom certificates](https://devcenter.heroku.com/articles/customizing-the-jdk#adding-custom-certificates)

You may also need to add custom certificates to the JDK’s cacerts. You may start with the keystore in your local JDK or [download the base Heroku keystore](https://heroku-java-cacerts-download.herokuapp.com/). Add the custom certificate with:

keytool -import -keystore cacerts -file custom.cer

You may be prompted for a password. The default password is changeit. You may then include the keystore in the slug by placing it in the .jdk-overlay/jre/lib/security/ directory of your app’s repository (or .jdk-overlay/lib/security/ for Java 9 and higher).

Now add any custom files to your Git repo like this:

git add .jdk-overlay

git commit -m "Custom JDK Files"

Then deploy your application, with the custom files, to Heroku:

git push heroku master

This overrides the keystore of your JDK. Your application no longer benefits from automatic keystore updates when the [Stack](https://devcenter.heroku.com/articles/stack) is updated.

## [Verify the copy](https://devcenter.heroku.com/articles/customizing-the-jdk#verify-the-copy)

The copies can be verified by starting a bash session on Heroku and checking the JDK directory. The JDK directory is located in $HOME/.jdk/.

For example, to verify custom certificates were copied correctly, the $HOME/.jdk/jre/lib/security/ directory can be checked.

heroku run bash

Running `bash` attached to terminal... up, run.1

~ $ keytool -list -keystore .jdk/jre/lib/security/cacerts

...

Your keystore contains 140 entries

...

## [Other examples](https://devcenter.heroku.com/articles/customizing-the-jdk#other-examples)

This method can be used for [Java extensions](http://docs.oracle.com/javase/tutorial/ext/basics/index.html) when necessary. Though a dependency management tool, such as Maven, should be the preferred mechanism for introducing dependencies.

### [Keep reading](https://devcenter.heroku.com/articles/customizing-the-jdk#keep-reading)

* [Java Advanced Topics](https://devcenter.heroku.com/categories/java-advanced-topics)

### [Feedback](https://devcenter.heroku.com/articles/customizing-the-jdk#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fcustomizing-the-jdk&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Reducing the Slug Size of Java Applications

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/reducing-the-slug-size-of-java-applications)

Last updated May 15, 2023

## Table of Contents

* [Using the Maven Clean plugin to remove build artifacts](https://devcenter.heroku.com/articles/reducing-the-slug-size-of-java-applications#using-the-maven-clean-plugin-to-remove-build-artifacts)
* [Using doLast in Gradle](https://devcenter.heroku.com/articles/reducing-the-slug-size-of-java-applications#using-dolast-in-gradle)
* [Excluding test dependencies](https://devcenter.heroku.com/articles/reducing-the-slug-size-of-java-applications#excluding-test-dependencies)
* [Adding a .slugignore file](https://devcenter.heroku.com/articles/reducing-the-slug-size-of-java-applications#adding-a-slugignore-file)
* [Using the Heroku Maven plugin](https://devcenter.heroku.com/articles/reducing-the-slug-size-of-java-applications#using-the-heroku-maven-plugin)

When deploying with Git, it’s common for very large Java applications to exceed the [slug size limits](https://devcenter.heroku.com/articles/slug-compiler) of the platform. If this happens, you’ll see an error like this when you deploy:

remote: -----> Compressing...

remote: ! Compiled slug size 583.3MB is too large, max is 500 MB.

There are a number of ways to mitigate this problem, including switching to [non-Git deployment](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin). In this article, you’ll learn how to do this and how to configure your application to reduce it’s slug size.

## [Using the Maven Clean plugin to remove build artifacts](https://devcenter.heroku.com/articles/reducing-the-slug-size-of-java-applications#using-the-maven-clean-plugin-to-remove-build-artifacts)

The Maven Clean plugin can be configured to remove non-essential artifacts after the build is complete. For example, if your application is packaged into a WAR file, and you would like remove all JAR files (because they are duplicated inside the WAR file), then you could configure the plugin like this in your pom.xml:

<plugin>

<artifactId>maven-clean-plugin</artifactId>

<version>3.0.0</version>

<executions>

<execution>

<id>clean-jar-artifacts</id>

<phase>install</phase>

<goals><goal>clean</goal></goals>

<configuration>

<excludeDefaultDirectories>true</excludeDefaultDirectories>

<filesets>

<fileset>

<directory>target/dependency/\*.jar</directory>

</fileset>

</filesets>

</configuration>

</execution>

</executions>

</plugin>

This removes all target/dependency/\*.jar files from the slug after the install phase.

Other common patterns for the <configuration> element include removing everything in the target/ directory except the WAR file:

<fileset>

<directory>target</directory>

<excludes>

<exclude>\*.war</exclude>

</excludes>

</fileset>

Or removing Node.js dependencies that were only needed to precompile JavaScript assets:

<fileset>

<directory>node\_modules</directory>

</fileset>

<fileset>

<directory>.heroku/node</directory>

</fileset>

This configuration is particularly common when the Java buildpack is combined with the Node.js buildpack to perform JavaScript optimization during the build.

## [Using doLast in Gradle](https://devcenter.heroku.com/articles/reducing-the-slug-size-of-java-applications#using-dolast-in-gradle)

For [Gradle deployments](https://devcenter.heroku.com/articles/deploying-gradle-apps-on-heroku), you can create a stage task with a [doLast](https://docs.gradle.org/current/userguide/tutorial_using_tasks.html) directive similar to this:

task stage {

dependsOn build

doLast {

delete fileTree(dir: "build", exclude: "libs")

delete fileTree(dir: "build/libs", exclude: "\*.jar")

}

}

The files you remove will depend on your application.

## [Excluding test dependencies](https://devcenter.heroku.com/articles/reducing-the-slug-size-of-java-applications#excluding-test-dependencies)

If you are using the maven-dependency-plugin to copy JAR files into the slug at build time, you may want to exclude your test dependencies. You can do this by setting the <includeScope> value in the plugin’s configuration:

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-dependency-plugin</artifactId>

<version>2.4</version>

<executions>

<execution>

<id>copy-dependencies</id>

<phase>package</phase>

<goals><goal>copy-dependencies</goal></goals>

<configuration>

<includeScope>compile</includeScope>

</configuration>

</execution>

</executions>

</plugin>

The included scopes defaults to test, which includes your test dependencies, so compile will actually exclude them.

## [Adding a .slugignore file](https://devcenter.heroku.com/articles/reducing-the-slug-size-of-java-applications#adding-a-slugignore-file)

If your Git repository includes binaries or other large files that are not needed at runtime, you can exclude them from the slug by [creating a .slugignore file](https://devcenter.heroku.com/articles/slug-compiler#ignoring-files-with-slugignore) in the root of your Git repo.

It is very common for the Git repository to include large files that are only needed when running tests. In this case, the .slugignore file might look like this:

\*.psd

\*.pdf

/test

You can inspect the extracted contents of your slug by running heroku run bash and using commands such as ls and du. This may help you identify large files that need to be added to the .slugignore file. Exactly what should be excluded depends on the needs of each application.

## [Using the Heroku Maven plugin](https://devcenter.heroku.com/articles/reducing-the-slug-size-of-java-applications#using-the-heroku-maven-plugin)

The [heroku-maven-plugin](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin) avoids Git deployment altogether and builds the slug locally before pushing it to Heroku. This allows it to package only what is absolutely necessary for the application to run.

If you need to include additional directories in the slug, you can do so with the <includes> element. For example:

<includes>

<include>etc/readme.txt</include>

</includes>

If you implement these suggestions, and still have a slug that is too large, please contact [Heroku support](https://devcenter.heroku.com/articles/support-channels).

### [Keep reading](https://devcenter.heroku.com/articles/reducing-the-slug-size-of-java-applications#keep-reading)

* [Java Advanced Topics](https://devcenter.heroku.com/categories/java-advanced-topics)

### [Feedback](https://devcenter.heroku.com/articles/reducing-the-slug-size-of-java-applications#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Freducing-the-slug-size-of-java-applications&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Deploying Tomcat-based Java Web Applications with Webapp Runner

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/java-webapp-runner)

Last updated October 09, 2023

## Table of Contents

* [Prerequisites](https://devcenter.heroku.com/articles/java-webapp-runner#prerequisites)
* [Create an application if you don’t already have one](https://devcenter.heroku.com/articles/java-webapp-runner#create-an-application-if-you-don-t-already-have-one)
* [Configure Maven to download Webapp Runner](https://devcenter.heroku.com/articles/java-webapp-runner#configure-maven-to-download-webapp-runner)
* [Run your application](https://devcenter.heroku.com/articles/java-webapp-runner#run-your-application)
* [Deploy your application to Heroku](https://devcenter.heroku.com/articles/java-webapp-runner#deploy-your-application-to-heroku)
* [Use distributed HTTP sessions with Memcache](https://devcenter.heroku.com/articles/java-webapp-runner#use-distributed-http-sessions-with-memcache)
* [Clone the source](https://devcenter.heroku.com/articles/java-webapp-runner#clone-the-source)

Webapp Runner allows you to launch an application in a Tomcat container on any computer that has a JRE installed. No previous steps to install Tomcat are required when using Webapp Runner. It’s a regular JAR file that can be executed and configured using the java command.

This article will walk you through building an application that launches with Webapp Runner and deploying that application to Heroku.

Follow each step to build an app from scratch, or skip to the end get the source for this article. You can also use almost any existing Maven webapp project.

## [Prerequisites](https://devcenter.heroku.com/articles/java-webapp-runner#prerequisites)

* Basic Java knowledge, including an installed version of the JVM and Maven.
* Basic Git knowledge, including an installed version of Git.

### [How does Webapp Runner work?](https://devcenter.heroku.com/articles/java-webapp-runner#how-does-webapp-runner-work)

When using Webapp Runner you’ll launch your application locally and on Heroku with a command like this:

java -jar webapp-runner.jar application.war

deploying app from: /Users/johnsimone/dev/gitrepos/devcenter-webapp-runner/target/webappRunnerSample.war

Feb 14, 2012 5:21:44 PM org.apache.coyote.AbstractProtocol init

INFO: Initializing ProtocolHandler ["http-bio-8080"]

Feb 14, 2012 5:21:44 PM org.apache.catalina.core.StandardService startInternal

INFO: Starting service Tomcat

Feb 14, 2012 5:21:44 PM org.apache.catalina.core.StandardEngine startInternal

INFO: Starting Servlet Engine: Apache Tomcat/8.0.30

Feb 14, 2012 5:21:44 PM org.apache.catalina.startup.ContextConfig webConfig

INFO: No global web.xml found

Feb 14, 2012 5:21:44 PM org.apache.coyote.AbstractProtocol start

INFO: Starting ProtocolHandler ["http-bio-8080"]

Webapp Runner will then launch a Tomcat instance with the given war deployed to it. This takes advantage of Tomcat’s embedded APIs. Webapp Runner is [open source](http://github.com/heroku/webapp-runner) so you can view or contribute to the source code.

## [Create an application if you don’t already have one](https://devcenter.heroku.com/articles/java-webapp-runner#create-an-application-if-you-don-t-already-have-one)

mvn archetype:generate -DarchetypeArtifactId=maven-archetype-webapp

...

[INFO] Generating project in Interactive mode

Define value for property 'groupId': : com.example

Define value for property 'artifactId': : helloworld

(you can pick any groupId or artifactId). You now have a complete Java web app in the helloworld directory.

## [Configure Maven to download Webapp Runner](https://devcenter.heroku.com/articles/java-webapp-runner#configure-maven-to-download-webapp-runner)

Although not necessary for using Webapp Runner it’s a good idea to have your build tool download Webapp Runner for you since your application will need it to run. You could, of course, just download Webapp Runner and use it to launch your application without doing this. However having all of your dependencies defined in your build descriptor is important for application portability and repeatability of deployment. In this case we’re using Maven so we’ll use the dependency plugin to download the jar. Add the following plugin configuration to your pom.xml:

<build>

...

<plugins>

...

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-dependency-plugin</artifactId>

<executions>

<execution>

<phase>package</phase>

<goals><goal>copy</goal></goals>

<configuration>

<artifactItems>

<artifactItem>

<groupId>com.heroku</groupId>

<artifactId>webapp-runner</artifactId>

<version>9.0.80.0</version>

<destFileName>webapp-runner.jar</destFileName>

</artifactItem>

</artifactItems>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

The version of Webapp Runner is pinned to the version of the underlying Tomcat server. Thus, [version 9.0.80.0 of Webapp Runner](http://search.maven.org/#artifactdetails%7Ccom.heroku%7Cwebapp-runner%7C9.0.78.0%7Cjar) uses version 9.0.80 of Tomcat.

## [Run your application](https://devcenter.heroku.com/articles/java-webapp-runner#run-your-application)

To build your application simply run:

mvn package

And then run your app using the java command:

java -jar target/dependency/webapp-runner.jar target/\*.war

That’s it. Your application should start up on port 8080.

Note: if you need your WAR file to be expanded before launching you can add the --expand-war option before target/\*.war

## [Deploy your application to Heroku](https://devcenter.heroku.com/articles/java-webapp-runner#deploy-your-application-to-heroku)

### [Create a Procfile](https://devcenter.heroku.com/articles/java-webapp-runner#create-a-procfile)

You declare how you want your application executed in Procfile in the project root. Create this file with a single line:

web: java $JAVA\_OPTS -jar target/dependency/webapp-runner.jar --port $PORT target/\*.war

Webapp Runner serves the WAR file at the root path (i.e. the URL won’t include the WAR file name in the path). If you need to change this, use the --path option. For more information on Webapp Runner options, [run the JAR file with the --help option](https://github.com/heroku/webapp-runner#options).

### [Deploy to Heroku](https://devcenter.heroku.com/articles/java-webapp-runner#deploy-to-heroku)

Commit your changes to Git:

git init

git add .

git commit -m "Ready to deploy"

Create the app:

heroku create

Creating high-lightning-129... done, stack is heroku-18

http://high-lightning-129.herokuapp.com/ | git@heroku.com:high-lightning-129.git

Git remote heroku added

Deploy your code:

git push heroku main

Counting objects: 227, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (117/117), done.

Writing objects: 100% (227/227), 101.06 KiB, done.

Total 227 (delta 99), reused 220 (delta 98)

-----> Heroku receiving push

-----> Java app detected

-----> Installing Maven 3.0.3..... done

-----> executing .maven/bin/mvn -B -Duser.home=/tmp/build\_1jems2so86ck4 -DskipTests=true clean install

[INFO] Scanning for projects...

[INFO]

[INFO] ------------------------------------------------------------------------

[INFO] Building webappRunnerSample Maven Webapp 1.0-SNAPSHOT

[INFO] ------------------------------------------------------------------------

...

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 36.612s

[INFO] Finished at: Tue Aug 30 04:03:02 UTC 2011

[INFO] Final Memory: 19M/287M

[INFO] ------------------------------------------------------------------------

-----> Discovering process types

Procfile declares types -> web

-----> Compiled slug size is 4.5MB

-----> Launching... done, v5

http://pure-window-800.herokuapp.com deployed to Heroku

Congratulations! Your web app should now be up and running on Heroku. Open it in your browser with:

heroku open

### [Deploying with the Heroku Maven Plugin](https://devcenter.heroku.com/articles/java-webapp-runner#deploying-with-the-heroku-maven-plugin)

In lieu of Git deployment, you may use the [Heroku Maven Plugin](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin) to deploy applications with webapp-runner. The setup for the Maven plugin is similar to what is described in this article, but there are a few key differences. In your pom.xml you will need to replace the maven-dependency-plugin with the heroku-maven-plugin and provide the proper configuration. Then, instead of deploying with a git push you will deploy with a mvn heroku:deploy-war command.

## [Use distributed HTTP sessions with Memcache](https://devcenter.heroku.com/articles/java-webapp-runner#use-distributed-http-sessions-with-memcache)

Explicitly storing session state in a database or other backend data store is a more scalable alternative to using distributed HTTP sessions.

Webapp runner supports the memcached-session-manager for Tomcat. In order to enable memcache backed sessions you need to make the configuration for your memcache instance available through environment variables and then enable the sesssion manager.

### [Make memcache configuration information available](https://devcenter.heroku.com/articles/java-webapp-runner#make-memcache-configuration-information-available)

The Heroku [Memcachier Add On](https://elements.heroku.com/addons/memcachier) will set the required environment variables for you. Once you have an existing app get the add on by running:

heroku addons:create memcachier:dev

Note: you may have to [verify](https://devcenter.heroku.com/articles/account-verification) your account before you can add this add on.

When running locally you can either set up a local install of memcache or connect to the remote memcache service provisioned for you by the Heroku add on.

When used with webapp runner the memcache backed session manager looks for 3 environment variables: MEMCACHIER\_SERVERS, MEMCACHIER\_USERNAME, MEMCACHIER\_PASSWORD. You can set these to point to a local memcache install or connect to the remote memcache service provisioned for you by the Heroku add on by running heroku config and copying the values into local environment variables.

### [Enable memcached-session-manager](https://devcenter.heroku.com/articles/java-webapp-runner#enable-memcached-session-manager)

To enable memcache backed sessions with webapp runner you include the following flag: --session-store memcache

So if launching locally your command would now look like:

java -jar target/dependency/webapp-runner.jar --session-store memcache target/\*.war

Or your Procfile would look like:

web: java $JAVA\_OPTS -jar target/dependency/webapp-runner.jar --port $PORT --session-store memcache target/\*.war

## [Clone the source](https://devcenter.heroku.com/articles/java-webapp-runner#clone-the-source)

If you want to skip the creation steps you can clone the finished sample (without memcache backed session):

git clone https://github.com/heroku/devcenter-webapp-runner

### [Keep reading](https://devcenter.heroku.com/articles/java-webapp-runner#keep-reading)

* [Java Advanced Topics](https://devcenter.heroku.com/categories/java-advanced-topics)

### [Feedback](https://devcenter.heroku.com/articles/java-webapp-runner#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fjava-webapp-runner&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Run Non-web Java Dynos on Heroku

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/run-non-web-java-processes-on-heroku)

Last updated December 16, 2019

## Table of Contents

* [Prerequisites](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#prerequisites)
* [Sample application](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#sample-application)
* [Run locally](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#run-locally)
* [Deploy to Heroku](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#deploy-to-heroku-1)
* [Create a Procfile](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#create-a-procfile)
* [Deploy to Heroku](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#deploy-to-heroku-2)
* [Scaling worker processes](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#scaling-worker-processes)
* [One-off dynos](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#one-off-dynos)
* [Scheduling jobs](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#scheduling-jobs)

Some applications can benefit from splitting logic into multiple components:

1. A web process that is consumed by the end-user
2. One or more non-web processes to perform background and admin tasks.

A non-web process can be either:

1. A long running process in a [Worker](https://devcenter.heroku.com/articles/process-model#mapping-the-unix-process-model-to-web-apps) dyno, that is waiting on events (either on a database or from a message queue)
2. A command executed in a [one-off dyno](https://devcenter.heroku.com/articles/one-off-dynos), which can be invoked manually from the command line or from a service like the [Heroku Scheduler](https://elements.heroku.com/addons/scheduler)

## [Prerequisites](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#prerequisites)

* Basic Java knowledge, including an installed version of the JVM and Maven.
* Basic Git knowledge, including an installed version of Git.

## [Sample application](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#sample-application)

A simple app that demonstrates the one-off and worker process types can be created as two simple Java classes and a build file:

sampleapp/

pom.xml

src/

main/

java/

OneOffProcess.java

WorkerProcess.java

### [src/main/java/OneOffProcess.java](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#src-main-java-oneoffprocess-java)

public class OneOffProcess

{

public static void main(String[] args)

{

System.out.println("OneOffProcess executed.");

}

}

### [src/main/java/WorkerProcess.java](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#src-main-java-workerprocess-java)

public class WorkerProcess

{

public static void main(String[] args)

{

while(true) {

try {

Thread.sleep(1000);

} catch(InterruptedException e) {}

System.out.println("Worker process woke up");

}

}

}

### [pom.xml](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#pom-xml)

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>

<artifactId>herokujavasample</artifactId>

<version>1.0-SNAPSHOT</version>

<build>

<plugins>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>appassembler-maven-plugin</artifactId>

<version>1.1.1</version>

<configuration>

<assembleDirectory>target</assembleDirectory>

<programs>

<program>

<mainClass>WorkerProcess</mainClass>

<name>worker</name>

</program>

<program>

<mainClass>OneOffProcess</mainClass>

<name>oneoff</name>

</program>

</programs>

</configuration>

<executions>

<execution>

<phase>package</phase><goals><goal>assemble</goal></goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

The app assembler plugin generates a convenient launch script for starting your application. A single pom.xml can define multiple web, worker or admin processes.

You can clone this project from [GitHub](https://github.com/heroku/devcenter-java-worker).

## [Run locally](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#run-locally)

To build your application simply run:

mvn package

Run the worker with:

sh target/bin/worker

Worker process woke up

Worker process woke up

Worker process woke up

...

(use target\bin\worker.bat on Windows). Run the one-off process with:

sh target/bin/oneoff

OneOffProcess executed.

That’s it. You are now ready to deploy to Heroku.

## [Deploy to Heroku](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#deploy-to-heroku-1)

## [Create a Procfile](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#create-a-procfile)

You declare how you want your application executed in a [Procfile](https://devcenter.heroku.com/articles/procfile) in the project root. Create this file as below:

worker: sh target/bin/worker

There is no need to add commands that you want executed in a one-off dyno to a Procfile - the one-off dyno mechanism lets you specify the command when you launch the one-off dyno.

## [Deploy to Heroku](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#deploy-to-heroku-2)

Commit your changes to Git:

git init

git add .

git commit -m "Ready to deploy"

Create the app:

heroku create

Creating empty-fire-9222... done, stack is heroku-18

http://empty-fire-9222.herokuapp.com/ | git@heroku.com:empty-fire-9222.git

Git remote heroku added

Deploy your code:

git push heroku master

Counting objects: 66, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (31/31), done.

Writing objects: 100% (66/66), 15.74 KiB, done.

Total 66 (delta 10), reused 30 (delta 9)

-----> Heroku receiving push

-----> Java app detected

-----> Installing Maven 3.0.3..... done

-----> executing /app/tmp/repo.git/.cache/.Maven/bin/mvn -B -Duser.home=/tmp/build\_14lc6nws0m7oc -Dmaven.repo.local=/app/tmp/repo.git/.cache/.m2/repository -DskipTests=true clean install

[INFO] Scanning for projects...

[INFO]

[INFO] ------------------------------------------------------------------------

[INFO] Building herokujavaworker 1.0-SNAPSHOT

[INFO] ------------------------------------------------------------------------

...

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 3.513s

[INFO] Finished at: Mon Nov 28 15:44:32 UTC 2011

[INFO] Final Memory: 12M/490M

[INFO] ------------------------------------------------------------------------

-----> Discovering process types

Procfile declares types -> worker

-----> Compiled slug size is 12K

-----> Launching... done, v5

http://empty-fire-9222.herokuapp.com deployed to Heroku

## [Scaling worker processes](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#scaling-worker-processes)

You can now start and scale your worker dynos using a command like this:

heroku ps:scale worker=1

Scaling worker processes... done, now running 1

By scaling your workers to more than one dyno you can have more listeners and thereby consume and process more messages simultaneously. To look at the logs for your worker dyno, you can use the command:

heroku logs --tail

2011-12-14T00:52:26+00:00 heroku[slugc]: Slug compilation started

2011-12-14T00:52:54+00:00 heroku[web.1]: State changed from created to down

2011-12-14T00:52:55+00:00 heroku[slugc]: Slug compilation finished

2011-12-14T00:53:17+00:00 heroku[worker.1]: State changed from created to starting

2011-12-14T00:53:17+00:00 heroku[api]: Scale to worker=1 by jesper@heroku.com

2011-12-14T00:53:17+00:00 heroku[worker.1]: Starting process with command `sh target/bin/worker`

2011-12-14T00:53:18+00:00 heroku[worker.1]: State changed from starting to up

2011-12-14T00:53:19+00:00 app[worker.1]: Worker process woke up

2011-12-14T00:53:20+00:00 app[worker.1]: Worker process woke up

2011-12-14T00:53:21+00:00 app[worker.1]: Worker process woke up

## [One-off dynos](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#one-off-dynos)

If your process is a command you wish to run manually on an as needed basis, you can do so with a one-off dyno. Use the heroku run command to start a one-off dyno and execute the command:

heroku run "sh target/bin/oneoff"

Running sh target/bin/oneoff attached to terminal... up, run.1

OneOffProcess executed.

## [Scheduling jobs](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#scheduling-jobs)

Applications that need to execute jobs based on some interval of time or frequency can use the [Scheduler add-on](https://devcenter.heroku.com/articles/scheduler) or can define a [custom clock process](https://devcenter.heroku.com/articles/scheduled-jobs-custom-clock-processes) with a library like [Quartz](https://devcenter.heroku.com/articles/scheduled-jobs-custom-clock-processes-java-quartz-rabbitmq).

### [Scheduled jobs versus workers](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#scheduled-jobs-versus-workers)

Scheduling one-off processes is a good way to perform admin tasks such as clearing a cache or triggering the creation of a report that is sent over email. These types of events happen infrequently and don’t need to scale up or down.

Worker processes are good for processing work that is being queued up by a front-end web process or by other worker processes. The workload may vary depending on the traffic to your app and you can scale up the number of workers so you can perform more work in parallel. You can also use a worker process if you simply need to process events more frequently than every 10 minutes.

### [Keep reading](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#keep-reading)

* [Java Advanced Topics](https://devcenter.heroku.com/categories/java-advanced-topics)

### [Feedback](https://devcenter.heroku.com/articles/run-non-web-java-processes-on-heroku#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Frun-non-web-java-processes-on-heroku&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Deploying Spring Boot Applications to Heroku

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/deploying-spring-boot-apps-to-heroku)

Last updated April 04, 2023

## Table of Contents

* [Creating a Spring Boot app](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku#creating-a-spring-boot-app)
* [Preparing a Spring Boot app for Heroku](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku#preparing-a-spring-boot-app-for-heroku)
* [Connecting to a database](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku#connecting-to-a-database)
* [Customizing the boot command](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku#customizing-the-boot-command)
* [Next steps](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku#next-steps)

The Spring Boot model of deploying standalone applications is a great fit for Heroku. You can use either Maven or Gradle to deploy a Spring application on Heroku, but for this guide we’ll assume that you’re using Maven and have [Maven 3](http://maven.apache.org/download.html) installed on your machine.

To begin, create a [free Heroku account](https://signup.heroku.com/). Then download and install the Heroku CLI.

Download the Heroku CLI for...

Once installed, you can use the heroku command from the terminal to log in using the email address and password you used when creating your Heroku account:

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

## [Creating a Spring Boot app](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku#creating-a-spring-boot-app)

To create a new Spring Boot application, first install the Spring Boot CLI as described in the [Spring Boot documentation](https://docs.spring.io/spring-boot/docs/current/reference/html/cli-using-the-cli.html). This will add a spring command to your path.

You can also start with a working [sample app](https://github.com/heroku/java-getting-started) if you’d prefer.

Use the CLI to create a new application by running this command:

spring init --dependencies=web demo

Then move into the application directory:

cd demo

The application does not have any custom logic by default – it’s just an empty template. To add some behavior, open the src/main/java/com/example/demo/DemoApplication.java file and put the following code in it:

In file src/main/java/com/example/demo/DemoApplication.java write:

package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.\*;

import org.springframework.stereotype.\*;

@Controller

@SpringBootApplication

public class DemoApplication {

@RequestMapping("/")

@ResponseBody

String home() {

return "Hello World!";

}

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

This creates a simple request mapping that displayed “Hello World!” in the browser. You could run the application locally to confirm this, but we’ll jump straight to running it on Heroku.

## [Preparing a Spring Boot app for Heroku](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku#preparing-a-spring-boot-app-for-heroku)

Before you can deploy the app to Heroku, you’ll need to create a Git repository for the application and add all of the code to it by running these commands:

git init

git add .

git commit -m "first commit"

You’ll deploy the app by pushing this Git repo to Heroku. It’s also possible to deploy using the [Heroku Maven plugin](https://devcenter.heroku.com/articles/deploying-java-applications-with-the-heroku-maven-plugin), but this guide will focus on using Git and the Heroku CLI.

In order to deploy to Heroku, you’ll first need to provision a new Heroku app. Run this command:

heroku create

Creating app... done, tranquil-mountain-19785

https://tranquil-mountain-19785.herokuapp.com/ | https://git.heroku.com/tranquil-mountain-19785.git

This also creates a remote repository called heroku in your local git repo. Heroku generates a random name (in this case nameless-lake-8055) for your app. You can rename it later with the heroku apps:rename command.

Now deploy your code:

git push heroku main

remote: Compressing source files... done.

remote: Building source:

remote:

remote: -----> Java app detected

remote: -----> Installing JDK 1.8... done

remote: -----> Executing Maven

...

remote: [INFO] BUILD SUCCESS

remote: [INFO] ------------------------------------------------------------------------

remote: [INFO] Total time: 15.129 s

remote: [INFO] Finished at: 2020-05-20T09:17:47Z

remote: [INFO] ------------------------------------------------------------------------

remote: -----> Discovering process types

remote: Procfile declares types -> (none)

remote: Default types for buildpack -> web

remote:

remote: -----> Compressing...

remote: Done: 65M

remote: -----> Launching...

remote: Released v3

remote: https://tranquil-mountain-19785.herokuapp.com/ deployed to Heroku

remote:

remote: Verifying deploy... done.

To https://git.heroku.com/tranquil-mountain-19785.git

\* [new branch] main -> main

Heroku automatically detects the application as a Maven/Java app due to the presence of a pom.xml file. It installed Java 8 by default, but you can easily configure this with a system.properties file as described in the [Specifying a Java version](https://devcenter.heroku.com/articles/java-support#specifying-a-java-version) Dev Center article. It will run your app using the [default command](https://devcenter.heroku.com/articles/java-support#default-web-process-type).

All that said, the application is now deployed. You can visit the app’s URL by running this command:

heroku open

You’ll see the “Hello World!” text in the browser.

You can view the logs for the application by running this command:

heroku logs --tail

2020-05-20T09:18:00.899237+00:00 app[web.1]: 2020-05-20 09:18:00.899 INFO 4 --- [ main] com.example.demo.DemoApplication : No active profile set, falling back to default profiles: default

2020-05-20T09:18:02.348822+00:00 app[web.1]: 2020-05-20 09:18:02.348 INFO 4 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 40371 (http)

2020-05-20T09:18:02.365045+00:00 app[web.1]: 2020-05-20 09:18:02.364 INFO 4 --- [ main] o.apache.catalina.core.StandardService : Starting service [Tomcat]

2020-05-20T09:18:02.365316+00:00 app[web.1]: 2020-05-20 09:18:02.365 INFO 4 --- [ main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.35]

2020-05-20T09:18:02.457002+00:00 app[web.1]: 2020-05-20 09:18:02.456 INFO 4 --- [ main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext

2020-05-20T09:18:02.457162+00:00 app[web.1]: 2020-05-20 09:18:02.457 INFO 4 --- [ main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1486 ms

2020-05-20T09:18:02.692238+00:00 app[web.1]: 2020-05-20 09:18:02.691 INFO 4 --- [ main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'

2020-05-20T09:18:02.891409+00:00 app[web.1]: 2020-05-20 09:18:02.891 INFO 4 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 40371 (http) with context path ''

2020-05-20T09:18:02.903573+00:00 app[web.1]: 2020-05-20 09:18:02.903 INFO 4 --- [ main] com.example.demo.DemoApplication : Started DemoApplication in 2.739 seconds (JVM running for 3.302)

2020-05-20T09:18:03.300746+00:00 heroku[web.1]: State changed from starting to up

Reload your application in the browser, and you’ll see another log message generated for that request. Press Control+C to stop streaming the logs.

To learn more about the basics of deploying a Maven-based Java application on Heroku, try following the [Getting Started with Java on Heroku](https://devcenter.heroku.com/articles/getting-started-with-java) guide. This guide covers many steps that are not specific to Spring Boot.

The remainder of this article provides a cursory overview of some of the most common settings you’ll need to adjust.

## [Connecting to a database](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku#connecting-to-a-database)

You can attach a PostgreSQL database to your app by running the following command from the CLI:

heroku addons:create heroku-postgresql

If you prefer to use MySQL or another database vendor, check out [Add-ons Marketplace](https://elements.heroku.com/addons) to see what is available.

Now you can list the configuration variables for your app to display the URL needed to connect to the database, DATABASE\_URL:

heroku config

=== tranquil-mountain-19785 Config Vars

DATABASE\_URL: postgres://user:password@ec2-1-2-3.compute-1.amazonaws.com:5432/databasename

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

heroku pg

=== DATABASE\_URL

Plan: Mini

Status: Available

Connections: 0/20

PG Version: 14.2

Created: 2020-05-20 09:18 UTC

Data Size: 7.9 MB

Tables: 0

Rows: 0/10000 (In compliance)

Fork/Follow: Unsupported

Rollback: Unsupported

Continuous Protection: Off

Add-on: postgresql-animated-55555

This indicates a Mini database is running Postgres 14.2, with a single row of data.

Once the database add-on has been created, Heroku will automatically populate the environment variables SPRING\_DATASOURCE\_URL, SPRING\_DATASOURCE\_USERNAME, and SPRING\_DATASOURCE\_PASSWORD. These environment variables should allow your Spring Boot application to connect to the database without any other configuration as long as you add a PostgreSQL JDBC driver to your dependencies like so:

In file pom.xml, on line 26 add:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

</dependency>

You can customize your application’s database configuration in your application.properties. For example:

In file src/main/resources/application.properties write:

spring.datasource.driverClassName=org.postgresql.Driver

spring.datasource.maxActive=10

spring.datasource.maxIdle=5

spring.datasource.minIdle=2

spring.datasource.initialSize=5

spring.datasource.removeAbandoned=true

Then you can then add a configuration bean to your app.

In file src/main/java/demo/DatabaseConfig.java write:

package com.example.demo;

import com.zaxxer.hikari.\*;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.context.annotation.\*;

import javax.sql.DataSource;

@Configuration

public class DatabaseConfig {

@Value("${spring.datasource.url}")

private String dbUrl;

@Bean

public DataSource dataSource() {

HikariConfig config = new HikariConfig();

config.setJdbcUrl(dbUrl);

return new HikariDataSource(config);

}

}

For more information, see [Connecting to Relational Databases on Heroku with Java](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java).

Now your application should be able to connect to the database. You can follow our guide on [Running Database Migrations for Java Apps](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps) to initialize the database. The [sample app](https://github.com/kissaten/spring-boot-heroku-demo) uses Liquibase.

## [Customizing the boot command](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku#customizing-the-boot-command)

You can override the default command used to run your app or define custom process types using a [Procfile](https://devcenter.heroku.com/articles/procfile). The correct command depends on what you need to do with your app. Common process types are used to [run a web process](https://devcenter.heroku.com/articles/java-support#default-web-process-type) or [run database migrations](https://devcenter.heroku.com/articles/running-database-migrations-for-java-apps).

## [Next steps](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku#next-steps)

For more complete examples of Spring Boot apps that run on Heroku see:

* [Getting Started on Heroku with Java](https://devcenter.heroku.com/articles/getting-started-with-java#introduction)
* [Spring Petclinic Demo for Heroku](https://github.com/kissaten/spring-petclinic)

Heroku provides a wide range of features for Spring applications. You can provision add-ons that introduce third-party cloud services like persistence, logging, monitoring and more. The [add-on marketplace](https://elements.heroku.com/addons/#data-stores) has a large number of data stores, from Redis and MongoDB providers, to Postgres and MySQL.

You can read more about [How Heroku Works](https://devcenter.heroku.com/articles/how-heroku-works) to get a technical overview of the concepts you’ll encounter while writing, configuring, deploying and running applications. Then visit the [Java category on Dev Center](https://devcenter.heroku.com/categories/java-support) to learn more about developing and deploying Spring applications. If you experience any trouble with your application as you migrate to Heroku, reach out to any of our [Support channels](https://devcenter.heroku.com/articles/support-channels).

For more information on deploying Spring apps, see the [Spring documentation](https://spring.io/quickstart) and the [Spring Boot documentation on deploy to Heroku](http://docs.spring.io/spring-boot/docs/current/reference/html/cloud-deployment.html#cloud-deployment-heroku).

### [Keep reading](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku#keep-reading)

* [Working with Spring Boot](https://devcenter.heroku.com/categories/working-with-spring-boot)

### [Feedback](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fdeploying-spring-boot-apps-to-heroku&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Preparing a Spring Boot App for Production on Heroku

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/preparing-a-spring-boot-app-for-production-on-heroku)

Last updated June 14, 2023

## Table of Contents

* [Force the use of HTTPS](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#force-the-use-of-https)
* [Rate-limit API calls](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#rate-limit-api-calls)
* [Use structured logging](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#use-structured-logging)
* [Use UUIDs for all Postgres Primary Keys](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#use-uuids-for-all-postgres-primary-keys)
* [Remove secrets from source code](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#remove-secrets-from-source-code)
* [Use a distributed session store](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#use-a-distributed-session-store)
* [Enable JVM Runtime Metrics](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#enable-jvm-runtime-metrics)
* [Configure alerting](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#configure-alerting)
* [Configure error and maintenance pages](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#configure-error-and-maintenance-pages)
* [Attach a logging add-on](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#attach-a-logging-add-on)
* [Attach an error tracking add-on](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#attach-an-error-tracking-add-on)
* [Attach a vulnerability detection add-on](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#attach-a-vulnerability-detection-add-on)
* [Further Reading](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#further-reading)

It is important to ensure that an app is secure, scalable, and resilient to failure before sending it to production. This guide provides an overview of many common and important steps required to make a Spring Boot app production-ready on Heroku.

## [Force the use of HTTPS](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#force-the-use-of-https)

Unless you have very specific needs, your app should be using HTTPS for all requests. Heroku provides an HTTPS URL (in the form https://<app-name>-<random-identifier>.herokuapp.com) for every app, as well as free tools for [adding your own domains](https://devcenter.heroku.com/articles/custom-domains) and [certificates](https://devcenter.heroku.com/articles/ssl).

You can enforce the use of HTTPS when your app is running on Heroku by adding the following configuration to your Spring Boot app.

@Configuration

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.requiresChannel()

.requestMatchers(r -> r.getHeader("X-Forwarded-Proto") != null)

.requiresSecure();

}

}

If you already have a WebSecurityConfigurerAdapter implementation, then add the above configuration to it.

This configuration tells Spring to redirect all plain HTTP requests back to the same URL using HTTPS if the X-Forwarded-Proto header is present. [Heroku sets the X-Forwarded-Proto header](https://devcenter.heroku.com/articles/http-routing#heroku-headers) for you, which means the request will be redirected back through the Heroku router where SSL is terminated. In your localhost environment, you can continue to use plain HTTP.

For more information see the official Spring Boot docs on how to [Enable HTTPS When Running behind a Proxy Server](https://docs.spring.io/spring-boot/docs/current/reference/html/howto-security.html#howto-enable-https).

## [Rate-limit API calls](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#rate-limit-api-calls)

Rate limiting is the process of controlling traffic to a server based on client IPs, blocked IPs, geolocation, and other factors. One of the most popular rate-limiting libraries for Java is [Bucket4j](https://github.com/vladimir-bukhtoyarov/bucket4j), which can be used with Spring Boot via the [Spring Boot Starter for Bucket4j](https://github.com/MarcGiffing/bucket4j-spring-boot-starter).

After adding the dependency and configuring a back-end, some minimal configuration might look like this in your application.yml:

bucket4j:

enabled: true

filters:

- cache-name: buckets

url: /\*

rate-limits:

- bandwidths:

- capacity: 5

time: 10

unit: seconds

This configuration limits an individual user to a maximum of 5 requests within a 10 second period.

## [Use structured logging](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#use-structured-logging)

Structured logging makes it easier to search and perform analytic operations on logging data. You can use [Logback](https://logback.qos.ch/) to encode your Spring Boot logs into JSON format by adding the following dependency to your app:

<dependency>

<groupId>net.logstash.logback</groupId>

<artifactId>logstash-logback-encoder</artifactId>

</dependency>

Then create a logback.xml configuration file in the src/main/resources directory with the following contents:

<configuration>

<appender name="consoleAppender" class="ch.qos.logback.core.ConsoleAppender">

<encoder class="net.logstash.logback.encoder.LogstashEncoder"/>

</appender>

<logger name="jsonLogger" additivity="false" level="DEBUG">

<appender-ref ref="consoleAppender"/>

</logger>

<root level="INFO">

<appender-ref ref="consoleAppender"/>

</root>

</configuration>

The result will be log lines that look like this:

{"@timestamp":"2018-04-16T09:36:34.641-05:00","@version":"1","message":"Started Main in 4.154 seconds (JVM running for 4.832)","logger\_name":"com.example.Main","thread\_name":"restartedMain","level":"INFO","level\_value":20000}

JSON formatted logs may be more difficult to read when using the heroku logs command, but they are far easier to search in a real production environment. Thus, you may choose to retain a human-readable log format in non-production environments.

## [Use UUIDs for all Postgres Primary Keys](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#use-uuids-for-all-postgres-primary-keys)

By default, Spring Data will use integers for primary keys in a Postgres database. However, this can leave your app vulnerable to exploits were attackers guess the next record identifier. It is preferable to use UUID primary keys, which cannot be guessed or enumerate. To configure your Spring Data models with UUID primary keys, use the @GeneratedValue annotation in conjunction with the java.util.UUID type like this:

@Entity

class MyModel {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private UUID id;

}

In this way, a random UUID is assigned when the model is created. Then add a UUID AttributeConverter to your app like this:

import javax.persistence.AttributeConverter;

import javax.persistence.Converter;

import java.util.UUID;

@Converter(autoApply = true)

public class UUIDConverter implements AttributeConverter<UUID, UUID> {

@Override

public UUID convertToDatabaseColumn(UUID attribute) {

return attribute;

}

@Override

public UUID convertToEntityAttribute(UUID dbData) {

return dbData;

}

}

UUIDs can be freely exposed without disclosing sensitive information. They are also unpredictable, perform well, and do not suffer from integer rollover.

## [Remove secrets from source code](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#remove-secrets-from-source-code)

It is unsafe to keep sensitive information in source code (i.e. checked into Git). Source code is stored in clear-text and is not encrypted on the server. Furthermore, storing secrets in source code increases the risk of accidentally connecting to a production service from a staging or test service.

All sensitive data (such as passwords, API tokens, and private keys) should be stored as configuration variables using the heroku config command. You can then access using environment variable substitution in your application.yml like this:

admin:

password: ${ADMIN\_PASSWORD}

Note that [most relational database URLs require zero-configuration on Heroku](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java) when used with Spring.

## [Use a distributed session store](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#use-a-distributed-session-store)

Storing sessions in-memory prohibits [disposability](https://12factor.net/disposability) and horizontal scalability of dynos. That’s why it’s important to use a distributed session store like Redis. Spring Boot has multiple mechanisms for storing sessions in Redis, including the [Spring Session module for Redis](https://spring.io/projects/spring-session-data-redis). However, Heroku recommends using [Redisson](https://github.com/redisson/redisson) as described in our article on [Java Session Handling](https://devcenter.heroku.com/articles/java-session-handling-on-heroku).

## [Enable JVM Runtime Metrics](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#enable-jvm-runtime-metrics)

The [JVM Runtime Metrics](https://devcenter.heroku.com/articles/language-runtime-metrics-jvm) feature allows you to view heap memory, non-heap memory, and garbage collection activity for any app that runs inside of the Java Virtual Machine (JVM). The feature is safe to use in production, and can help you identify performance related problems.

## [Configure alerting](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#configure-alerting)

When your application experiences a problem, it should alert a human. At a minimum, it should:

* Alert a human if it is down
* Alert a human if error rates go above specific thresholds
* Alert a human if latency is high

Heroku’s [Threshold Alerting](https://devcenter.heroku.com/articles/metrics#threshold-alerting) feature is available to apps running on Professional dynos. But you can also choose from the many [Alerting and Monitoring add-ons](https://elements.heroku.com/addons) in the Heroku Add-on Marketplace.

## [Configure error and maintenance pages](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#configure-error-and-maintenance-pages)

Heroku provides a mechanism for configuring the static HTML pages that are shown to users when your application experiences an error (such as crashing) or goes down for maintenance. Please see the Dev Center guide to [Customizing Error Pages](https://devcenter.heroku.com/articles/error-pages#customize-pages) for more information.

## [Attach a logging add-on](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#attach-a-logging-add-on)

By default, Heroku stores 1500 lines of logs from your application. However, it makes the full log stream available as a service that several add-on providers consume to provide features such as log persistence, search, and alerts via email or SMS.

You can provision one of these logging add-ons, Papertrail, by running the following command on your app:

heroku addons:create papertrail

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.

## [Attach an error tracking add-on](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#attach-an-error-tracking-add-on)

While your application logs capture the regular activity of your server processes, an error tracking add-on can capture more detail for exceptional cases. This can be useful in identifying common problems your users encounter, or to diagnose poor performance. The Add-on Marketplace has a number of options including the Rollbar service, which you can add to your app by running:

heroku addons:create rollbar

After following the guide for [setting up Rollbar integration](https://devcenter.heroku.com/articles/rollbar), you’ll begin to see a record of all errors with their associated stack traces and other details coming from your app.

## [Attach a vulnerability detection add-on](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#attach-a-vulnerability-detection-add-on)

The final add-on every production app should use is one that detects security vulnerabilities in your source code. One such service is [Snyk](https://devcenter.heroku.com/articles/snyk), which you can attach to your app by running:

heroku addons:create snyk

Snyk will scan your source code, and compare the dependencies in either your pom.xml or build.gradle against its database of known security vulnerabilities. If it finds any it will send a notification telling you how to update it.

## [Further Reading](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#further-reading)

For more information and guidance on running Spring Boot apps in production, please see the official Spring documentation on [Moving to Production](https://docs.spring.io/spring-boot/docs/current/reference/html/_moving_to_production.html)

### [Keep reading](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#keep-reading)

* [Working with Spring Boot](https://devcenter.heroku.com/categories/working-with-spring-boot)

### [Feedback](https://devcenter.heroku.com/articles/preparing-a-spring-boot-app-for-production-on-heroku#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fpreparing-a-spring-boot-app-for-production-on-heroku&utm_campaign=login&utm_medium=feedback&utm_source=web)

# Scaling a Spring Boot Application with Memcache

English — [日本語に切り替える](https://devcenter.heroku.com/ja/articles/spring-boot-memcache)

Last updated November 10, 2022

## Table of Contents

* [Prerequisites](https://devcenter.heroku.com/articles/spring-boot-memcache#prerequisites)
* [Deploying a Spring Boot application to Heroku](https://devcenter.heroku.com/articles/spring-boot-memcache#deploying-a-spring-boot-application-to-heroku)
* [Add task list functionality](https://devcenter.heroku.com/articles/spring-boot-memcache#add-task-list-functionality)
* [Add caching to Spring Boot](https://devcenter.heroku.com/articles/spring-boot-memcache#add-caching-to-spring-boot)
* [Further reading & resources](https://devcenter.heroku.com/articles/spring-boot-memcache#further-reading-resources)

Memcache is a technology that improves the performance and scalability of web apps and mobile app backends. You should consider using Memcache when your pages are loading too slowly or your app is having scalability issues. Even for small sites, Memcache can make page loads snappy and help future-proof your app.

This guide shows how to create a simple [Spring Boot 2](https://spring.io/projects/spring-boot) application (based on the [Spring Framework 5](https://spring.io/)), deploy it to Heroku, then add Memcache to alleviate a performance bottleneck.

You can view the [source code](https://github.com/memcachier/examples-spring-boot) or deploy it with this Heroku Button:

[Deploy to Heroku](https://heroku.com/deploy?template=http://github.com/memcachier/examples-spring-boot)

## [Prerequisites](https://devcenter.heroku.com/articles/spring-boot-memcache#prerequisites)

Before you complete the steps in this guide, make sure you have all of the following:

* Familiarity with Java (and ideally Spring Boot)
* A Heroku user account ([signup is free and instant](https://signup.heroku.com/signup/dc))
* Maven, and the [Heroku CLI](https://devcenter.heroku.com/articles/heroku-cli) installed on your computer

## [Deploying a Spring Boot application to Heroku](https://devcenter.heroku.com/articles/spring-boot-memcache#deploying-a-spring-boot-application-to-heroku)

To easily create a Spring Boot application we recommend you to install the [Spring Boot CLI](https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started-installing-spring-boot.html#getting-started-installing-the-cli). If you don’t want to install the CLI you can also configure and download a Spring Boot skeleton from the [Spring Initializer](http://start.spring.io/).

There is a ruby on rails application called spring as well. In case you have it installed or in case you have the Ruby version manager rbenv installed which shims this binary, create an alias for the spring cli (e.g., alias springboot='/opt/spring-boot-cli/bin/spring').

With the CLI you can easily create a skeleton:

spring init --d=web,data-jpa,thymeleaf -g com.memcachier -a tutorial -n TaskList memcached\_tutorial

cd memcached\_tutorial

The created skeleton is a web app with database support (data-jpa) and using the thymeleaf template language. Spring Boot also supports other template languages such as JSP, groovy, freemaker, and mustache.

### [Create a Heroku app](https://devcenter.heroku.com/articles/spring-boot-memcache#create-a-heroku-app)

Turning the Spring Boot skeleton into a Heroku app is easily done with 3 simple steps:

1. In order to let Heroku know how to start up your app, you need to add a [Procfile](https://devcenter.heroku.com/articles/procfile):

$ echo 'web: java -Dserver.port=$PORT $JAVA\_OPTS -jar target/\*.jar' > Procfile

1. Initialize a Git repository and commit the skeleton:

$ git init

$ git add .

$ git commit -m 'Spring Boot skeleton for Heroku'

1. Create a Heroku app:

$ heroku create

In addition to creating the actual Heroku application this command also adds the corresponding remote to your local Git repository.

## [Add task list functionality](https://devcenter.heroku.com/articles/spring-boot-memcache#add-task-list-functionality)

Let’s add a task list to the app that enables users to view, add, and delete tasks. To accomplish this, we need to:

1. Set up the database
2. Create a Task entity and a table to store them
3. Create the view and controller logic

### [Set up a PostgreSQL database](https://devcenter.heroku.com/articles/spring-boot-memcache#set-up-a-postgresql-database)

Before we can configure a database in Spring Boot, we need to create the database. On Heroku, you can add a free development database to your app like so:

heroku addons:create heroku-postgresql:mini

This creates a PostgreSQL database for your app and adds a DATABASE\_URL config var that contains its URL.

Spring Boot requires the variable SPRING\_DATASOURCE\_URL to be set. This variable contains the same URL as DATABASE\_URL except that it starts with jdbc:postgresql instead of postgres. Heroku will automatically populate this variable at runtime so you don’t have to worry about it.

To use this database we need to install a few packages. Add the following dependencies in pom.xml:

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

</dependency>

<dependency>

<groupId>javax.xml.bind</groupId>

<artifactId>jaxb-api</artifactId>

<version>2.3.0</version>

</dependency>

<dependency>

<groupId>org.liquibase</groupId>

<artifactId>liquibase-core</artifactId>

<version>3.6.1</version>

</dependency>

* The first dependency simply is the PostgreSQL driver.
* The second dependency just adds the JAXB APIs, as they are no longer available out of the box for newer Java SE versions. For more information see [this StackOverflow thread](https://stackoverflow.com/questions/43574426/how-to-resolve-java-lang-noclassdeffounderror-javax-xml-bind-jaxbexception-in-j).
* The third dependency allows you to create and run liquibase database migrations.

Now we can configure the database in src/main/resources/application.properties:

spring.datasource.driverClassName=org.postgresql.Driver

spring.datasource.maxActive=10

spring.datasource.maxIdle=5

spring.datasource.minIdle=2

spring.datasource.initialSize=5

spring.datasource.removeAbandoned=true

# Supress exception regarding missing PostgreSQL CLOB feature at Spring startup.

# See http://vkuzel.blogspot.ch/2016/03/spring-boot-jpa-hibernate-atomikos.html

spring.jpa.properties.hibernate.temp.use\_jdbc\_metadata\_defaults = false

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQL9Dialect

Your PostgreSQL database is now ready to be used. Save the changes with

git commit -am 'Database setup'

For more info on connecting to relational databases from Java on Heroku, see [this guide](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java).

### [Create the Task entity and database table](https://devcenter.heroku.com/articles/spring-boot-memcache#create-the-task-entity-and-database-table)

In order to create and store tasks we need to create three things: a Task entity, a repository to teach Spring Boot how to store and retrieve tasks, and a migration that creates the actual table in the database.

1. Add the Task entity to src/main/java/com/memcachier/tutorial/Task.java:

package com.memcachier.tutorial;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import org.hibernate.validator.constraints.NotEmpty;

@Entity

public class Task {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

private Long id;

@NotEmpty

private String name;

protected Task() {}

public Task(String name) {

this.name = name;

}

public Long getId() {

return this.id;

}

public String getName() {

return this.name;

}

public void setName(String name) {

this.name = name;

}

@Override

public String toString() {

return String.format("Task[id=%d, name='%s']", this.id, this.name);

}

}

1. Create a repository in src/main/java/com/memcachier/tutorial/TaskRepository.java:

package com.memcachier.tutorial;

import java.util.List;

import org.springframework.data.repository.CrudRepository;

public interface TaskRepository extends CrudRepository<Task, Long> {}

If you need more than basic CRUD functions to access your data you can also extend a PagingAndSortingRepository or a JpaRepository instead. See [this StackOverflow thread](https://stackoverflow.com/questions/14014086/what-is-difference-between-crudrepository-and-jparepository-interfaces-in-spring) for more information.

1. Create a liquibase migration in src/main/resources/db/changelog/db.changelog-master.yaml:

databaseChangeLog:

- changeSet:

id: 1

author: memcachier

changes:

- createTable:

tableName: task

columns:

- column:

name: id

type: int

autoIncrement: true

constraints:

primaryKey: true

nullable: false

- column:

name: name

type: varchar(255)

constraints:

nullable: false

Note, you will need to create the db and changelog folders. The migration will run automatically when the application starts.

Let’s save the changes so far:

git add .

git commit -m 'Task table setup'

### [Create the task list application](https://devcenter.heroku.com/articles/spring-boot-memcache#create-the-task-list-application)

The actual application consists of a view that is displayed in the frontend and a controller that implements the functionality in the backend.

* Create a controller in src/main/java/com/memcachier/tutorial/TaskController.java:

package com.memcachier.tutorial;

import javax.validation.Valid;

import java.lang.Iterable;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.ui.ModelMap;

import org.springframework.validation.BindingResult;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RequestParam;

@Controller

@RequestMapping("/")

public class TaskController {

private TaskRepository taskRepo;

@Autowired

public TaskController(TaskRepository repo) {

this.taskRepo = repo;

}

@RequestMapping(method = RequestMethod.GET)

public String showAllTasks(ModelMap model) {

Iterable<Task> tasks = this.taskRepo.findAll();

model.addAttribute("tasks", tasks);

model.addAttribute("newTask", new Task());

return "task";

}

@RequestMapping(method = RequestMethod.POST)

public String newTask(ModelMap model,

@ModelAttribute("newTask") @Valid Task task,

BindingResult result) {

if (!result.hasErrors()) {

this.taskRepo.save(task);

}

return showAllTasks(model);

}

@RequestMapping(method = RequestMethod.DELETE)

public String deleteTask(ModelMap model, @RequestParam("taskId") Long id) {

this.taskRepo.deleteById(id);

return showAllTasks(model);

}

}

This controller contains all functionality to GET all tasks and render the task view, to POST a new task that will then be saved to the database, and to DELETE existing tasks.

* Create a view in src/main/resources/templates/task.html:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<title>MemCachier Spring Boot Tutorial</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

<!-- Fonts -->

<link

href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.4.0/css/font-awesome.min.css"

rel="stylesheet"

type="text/css"

/>

<!-- Bootstrap CSS -->

<link

href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"

rel="stylesheet"

/>

</head>

<body>

<div class="container">

<!-- New Task Card -->

<div class="card">

<div class="card-body">

<h5 class="card-title">New Task</h5>

<form th:object="${newTask}" method="POST">

<div class="form-group">

<input

type="text"

class="form-control"

placeholder="Task Name"

th:field="\*{name}"

/>

</div>

<button type="submit" class="btn btn-default">

<i class="fa fa-plus"></i> Add Task

</button>

</form>

</div>

</div>

<!-- Current Tasks -->

<div th:if="${not #lists.isEmpty(tasks)}">

<div class="card">

<div class="card-body">

<h5 class="card-title">Current Tasks</h5>

<table class="table table-striped">

<tr th:each="task : ${tasks}">

<!-- Task Name -->

<td th:text="${task.name}" class="table-text"></td>

<!-- Delete Button -->

<td>

<form th:object="${deleteTask}" th:method="DELETE">

<input

type="hidden"

name="taskId"

th:value="${task.id}"

/>

<button type="submit" class="btn btn-danger">

<i class="fa fa-trash"></i> Delete

</button>

</form>

</td>

</tr>

</table>

</div>

</div>

</div>

</div>

<!-- Bootstrap related JavaScript -->

<script src="https://code.jquery.com/jquery-3.2.1.slim.min.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.js"></script>

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"></script>

</body>

</html>

The view basically consists of two cards, one that contains a form to create new tasks and another containing a table with the existing tasks and a delete button to remove the corresponding task.

Let us see what we have done so far and deploy the task list to Heroku:

git add .

git commit -m 'Add task list view and controller'

git push heroku master

heroku open

Test the application by adding a few tasks. We now have a functioning task list running on Heroku. With this complete, we can learn how to improve its performance with Memcache.

## [Add caching to Spring Boot](https://devcenter.heroku.com/articles/spring-boot-memcache#add-caching-to-spring-boot)

Memcache is an in-memory, distributed cache. Its primary API consists of two operations: SET(key, value) and GET(key). Memcache is like a hashmap (or dictionary) that is spread across multiple servers, where operations are still performed in constant time.

The most common use for Memcache is to cache expensive database queries and HTML renders so that these expensive operations don’t need to happen over and over again.

### [Set up Memcache](https://devcenter.heroku.com/articles/spring-boot-memcache#set-up-memcache)

To use Memcache in Spring Boot, you first need to provision an actual Memcache cache. You can easily get one for free with the [MemCachier add-on](https://elements.heroku.com/addons/memcachier):

heroku addons:create memcachier:dev

Then we need to configure the appropriate dependencies. We will use [simple-spring-memcached](https://github.com/ragnor/simple-spring-memcached) with [XMemcached](https://github.com/killme2008/xmemcached/) to use Memcache within Spring Boot. You can use also simple-spring-memcached with [SpyMemcached](https://github.com/couchbase/spymemcached). If you wish to do so, please refer to the [MemCachier documentation](https://devcenter.heroku.com/articles/memcachier#spring-boot).

To use simple-spring-memcached add the following to your pom.xml:

<dependency>

<groupId>com.google.code.simple-spring-memcached</groupId>

<artifactId>xmemcached-provider</artifactId>

<version>4.0.0</version>

</dependency>

<!-- Force XMemcached to version 2.4.3 simple-spring-memcached uses 2.4.0 -->

<dependency>

<groupId>com.googlecode.xmemcached</groupId>

<artifactId>xmemcached</artifactId>

<version>2.4.3</version>

</dependency>

Now we can configure Memcache for Spring in src/main/java/com/memcachier/tutorial/MemCachierConfig.java:

package com.memcachier.tutorial;

import java.net.InetSocketAddress;

import java.util.List;

import java.util.Map;

import java.util.HashMap;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import com.google.code.ssm.CacheFactory;

import com.google.code.ssm.config.AbstractSSMConfiguration;

import com.google.code.ssm.config.DefaultAddressProvider;

import com.google.code.ssm.providers.xmemcached.XMemcachedConfiguration;

import com.google.code.ssm.providers.xmemcached.MemcacheClientFactoryImpl;

import net.rubyeye.xmemcached.auth.AuthInfo;

import net.rubyeye.xmemcached.utils.AddrUtil;

@Configuration

public class MemCachierConfig extends AbstractSSMConfiguration {

@Bean

@Override

public CacheFactory defaultMemcachedClient() {

String serverString = System.getenv("MEMCACHIER\_SERVERS").replace(",", " ");

List<InetSocketAddress> servers = AddrUtil.getAddresses(serverString);

AuthInfo authInfo = AuthInfo.plain(System.getenv("MEMCACHIER\_USERNAME"),

System.getenv("MEMCACHIER\_PASSWORD"));

Map<InetSocketAddress, AuthInfo> authInfoMap =

new HashMap<InetSocketAddress, AuthInfo>();

for(InetSocketAddress server : servers) {

authInfoMap.put(server, authInfo);

}

final XMemcachedConfiguration conf = new XMemcachedConfiguration();

conf.setUseBinaryProtocol(true);

conf.setAuthInfoMap(authInfoMap);

final CacheFactory cf = new CacheFactory();

cf.setCacheClientFactory(new MemcacheClientFactoryImpl());

cf.setAddressProvider(new DefaultAddressProvider(serverString));

cf.setConfiguration(conf);

return cf;

}

}

This configures simple-spring-memcached which allows you to use its caching annotations. Spring also provides built in caching annotations that can be enabled via simple-spring-memcached. However, in this tutorial we will use the annotations provided simple-spring-memcached because they are generally more flexible and better suited for a Memcached backed cache. Nevertheless, this tutorial would work just as well with Spring’s annotations. If you prefer to use Spring’s built in caching annotations, please refer to the [MemCachier documentation](https://devcenter.heroku.com/articles/memcachier/#spring-boot).

### [Cache expensive database queries](https://devcenter.heroku.com/articles/spring-boot-memcache#cache-expensive-database-queries)

Memcache is often used to cache expensive database queries. In this simple example we do not have any expensive queries but for the sake of learning, let’s assume that getting all tasks from the database is an expensive operation.

To cache the Task queries we will extend the TaskRepository with methods that implement caching. Extending a repository in Spring Boot involves three steps:

1. Build an interface with the methods that should be added to the TaskRepository in src/main/java/com/memcachier/tutorial/CachedTaskRepository.java:

package com.memcachier.tutorial;

import java.lang.Iterable;

public interface CachedTaskRepository {

public Iterable<Task> findAllCached();

}

1. Create a an implementation for this interface in src/main/java/com/memcachier/tutorial/TaskRepositoryImpl.java:

package com.memcachier.tutorial;

import java.lang.Iterable;

import org.springframework.beans.factory.annotation.Autowired;

import com.google.code.ssm.api.ReadThroughAssignCache;

public class TaskRepositoryImpl implements CachedTaskRepository {

@Autowired

TaskRepository taskRepository;

@ReadThroughAssignCache(namespace="Taskrepo", assignedKey="all")

public Iterable<Task> findAllCached() {

return this.taskRepository.findAll();

}

}

The filename of the implementation must follow the naming convention <REPOSITORY-NAME>Impl.java.

You can access the rest of the CRUD interface of the TaskRepository by just adding an @Autowired reference to it.

The caching occurs here via the @ReadThroughAssignCache annotation. All @ReadThrough\*Cache annotations do the following:

* Check if value is in cache and if true return said value.
* If not in cache, execute function, return its value and store said value in the cache.

The Assign version of this annotation will use an assigned key that is declared in the annotation. For more information about the annotations, refer to the [Simple Spring Memcached documentation](https://github.com/ragnor/simple-spring-memcached/wiki/Getting-Started#usage).

1. Make sure this implementation is integrated into the TaskRepository. This is simply done by making the TaskRepository interface also extend the CachedTaskRepository interface:

// ...

public interface TaskRepository extends CrudRepository<Task, Long>, CachedTaskRepository {}

A note on caching annotations: Spring uses proxies to handle caching annotations. For this reason you cannot create a private method inside your controller, add a caching annotation and expect the method to be cached. In simple terms, the cached method must be part of a component that is accessed via it’s interface. For more information see [this StackOverflow thread](https://stackoverflow.com/questions/12115996/spring-cache-cacheable-method-ignored-when-called-from-within-the-same-class) and the therein mentioned references.

Now we have the methods to cache all tasks but in order for them to work the Task data type in src/main/java/com/memcachier/tutorial/Task.java needs to be serializable:

// ...

import java.io.Serializable;

public class Task implements Serializable {

// ...

}

Finally, we can now get the cached tasks in the controller in src/main/java/com/memcachier/tutorial/TaskController.java:

// ...

public String showAllTasks(ModelMap model) {

Iterable<Task> tasks = this.taskRepo.findAllCached();

// ...

}

// ...

Let us deploy and test this new functionality:

git add .

git commit -m 'Add caching with MemCachier'

git push heroku master

heroku open

To see what is going on in your cache, open the MemCachier dashboard:

heroku addons:open memcachier

The first time you loaded your task list you should have gotten an increase for the get misses and the set commands. Every subsequent reload of the task list should increase the get hits (refresh the stats in the dashboard).

Our cache is working but there is still a mayor problem. Add a new task and see what happens. No new task appears on our current tasks list. The new task was created in our database but our app is serving the stale task list from the cache.

### [Clear stale data](https://devcenter.heroku.com/articles/spring-boot-memcache#clear-stale-data)

As important as caching data, is to invalidate it when it becomes stale. In our example the cached task list becomes stale whenever a new task is added or an existing task is removed. We need to make sure our cache is invalidated whenever one of these two actions are performed.

We can add wrappers to the save and delete methods in the TaskRepository that clear the cache with the following two steps:

1. Declare the wrapper methods in the CachedTaskRepository interface in src/main/java/com/memcachier/tutorial/CachedTaskRepository.java:

// ...

public interface CachedTaskRepository {

public Iterable<Task> findAllCached();

public Task saveAndClearCache(Task t);

public void deleteByIdAndClearCache(Long id);

}

1. Implement the wrapper methods in src/main/java/com/memcachier/tutorial/TaskRepositoryImpl.java:

// ...

import com.google.code.ssm.api.InvalidateAssignCache;

public class TaskRepositoryImpl implements CachedTaskRepository {

// ...

@InvalidateAssignCache(namespace="Taskrepo", assignedKey="all")

public Task saveAndClearCache(Task t){

return this.taskRepository.save(t);

}

@InvalidateAssignCache(namespace="Taskrepo", assignedKey="all")

public void deleteByIdAndClearCache(Long id){

this.taskRepository.deleteById(id);

}

}

The stale data is invalidated here via @InvalidateAssignCache annotation. Just as @ReadThroughAssignCache it acts on the assigned key that is declared in the annotation.

Now we can use these wrapper functions in our controller to clear the cache whenever a request comes in to add or delete a task. To do so replace save and deleteById in src/main/java/com/memcachier/tutorial/TaskController.java with saveAndClearCache and deleteByIdAndClearCache like so:

// ...

@RequestMapping(method = RequestMethod.POST)

public String newTask(ModelMap model,

@ModelAttribute("newTask") @Valid Task task,

BindingResult result) {

if (!result.hasErrors()) {

this.taskRepo.saveAndClearCache(task);

}

return showAllTasks(model);

}

@RequestMapping(method = RequestMethod.DELETE)

public String deleteTask(ModelMap model, @RequestParam("taskId") Long id) {

this.taskRepo.deleteByIdAndClearCache(id);

return showAllTasks(model);

}

Deploy the fixed task list:

git commit -am 'Clear stale data from cache'

git push heroku master

heroku open

Add a new task and you will see all the tasks appear you have added since we implemented caching for the task list.

## [Further reading & resources](https://devcenter.heroku.com/articles/spring-boot-memcache#further-reading-resources)

* [MemCachier Add-on Page](https://elements.heroku.com/addons/memcachier)
* [MemCachier Documentation](https://devcenter.heroku.com/articles/memcachier)
* [Advance Memcache Usage](https://devcenter.heroku.com/articles/advanced-memcache)
* [Heroku Spring Boot Guide](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku)
* [Simple Spring Memcached Documentation](https://github.com/ragnor/simple-spring-memcached/wiki/Getting-Started)
* [Spring Caching Guide](https://spring.io/guides/gs/caching/)
* [Spring Caching Documentation](https://docs.spring.io/spring/docs/5.0.5.RELEASE/spring-framework-reference/integration.html#cache)

### [Keep reading](https://devcenter.heroku.com/articles/spring-boot-memcache#keep-reading)

* [Working with Spring Boot](https://devcenter.heroku.com/categories/working-with-spring-boot)

### [Feedback](https://devcenter.heroku.com/articles/spring-boot-memcache#feedback)

[Log in to submit feedback.](https://devcenter.heroku.com/login?back_to=%2Farticles%2Fspring-boot-memcache&utm_campaign=login&utm_medium=feedback&utm_source=web)