

UPDATED FOR JHIPSTER 5.X



THE JHIPSTER MINI-BOOK

Matt Raible

InfoQ
Enterprise Software Development Series

THE JHIPSTER MINI-BOOK

Matt Raible

The JHipster Mini-Book

© 2018 Matt Raible. All rights reserved. Version 5.0.1.

Published by C4Media, publisher of InfoQ.com.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recoding, scanning or otherwise except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

Production Editor: Ana Ciobotaru

Copy Editor: Lawrence Nyveen

Cover and Interior Design: Dragos Balasoiu

Library of Congress Cataloguing-in-Publication Data: ISBN: 978-1-329-63814-3

Table of Contents

Dedication	1
Acknowledgements	2
Preface	3
What is in an InfoQ mini-book?	3
Who this book is for	4
What you need for this book	4
Conventions	4
Reader feedback	6
Introduction	7
Building an app with JHipster	8
Creating the application	10
Building the UI and business logic	16
Application improvements	26
Deploying to Heroku	58
Monitoring and analytics	64
Securing user data	65
Continuous integration and deployment	65
Code quality	71
Progressive web apps	72
Source code	74
Upgrading 21-Points Health	74
Summary	74
JHipster's UI components	76
Angular	77
Bootstrap	87
Internationalization (i18n)	100
Sass	101
Webpack	103
WebSockets	105
Browsersync	109
Summary	111
JHipster's API building blocks	112
Spring Boot	113
Spring WebFlux	126
Maven versus Gradle	126

IDE support: Running, debugging, and profiling	129
Security	130
JPA versus MongoDB versus Cassandra	132
Liquibase	134
Elasticsearch	135
Deployment	136
Summary	136
Microservices with JHipster	138
History of microservices	139
Why microservices?	140
Microservices with JHipster	141
Generate an API gateway and microservice applications	142
Start JHipster Registry, Keycloak, and MongoDB	147
Run your microservices architecture	148
Deploy to Heroku	157
Source code	163
Summary	163
Action!	164
Additional reading	164
About the author	165

Dedication

I dedicate this book to my parents, Joe and Barbara Raible. They raised my sister and me in the backwoods of Montana, with no electricity and no running water. We had fun-loving game nights, lots of walking, plenty of farm animals, an excellent garden, and a unique perspective on life.

Thanks, Mom and Dad - you rock!

Acknowledgements

I'm extremely grateful to my family, for putting up with my late nights and extended screen time while I worked on this book.

To Rod Johnson and Juergen Hoeller, thanks for inventing Spring and changing the lives of Java developers forever. To Phil Webb and Dave Syer, thanks for creating Spring Boot and breathing a breath of fresh air into the Spring Framework. Last but not least, thanks to Josh Long for first showing me Spring Boot and for being one of the most enthusiastic Spring developers I've ever met.

I'd like to thank this book's tech editors, Dennis Sharpe and Jeet Gajjar. Their real-world experience with JHipster made the code sections a lot more bulletproof.

This book's copy editor, Lawrence Nyveen, was a tremendous help in correcting my words and making this book easier to read. Thanks Laurie!

Finally, kudos to Julien Dubois and Deepu K Sasidharan for creating and improving JHipster. They've done a helluva job in turning it into a widely used and successful open-source project.

Preface

A few years ago, I consulted at several companies that used Spring and Java to develop their back-end systems. On those projects, I introduced Spring Boot to simplify development. DevOps teams often admired its external configuration and its starter dependencies made it easy to develop SOAP and REST APIs.

I used AngularJS for several years as well. For the first project I used AngularJS on, in early 2013, I implemented in 40% of the code that jQuery would've required. I helped that company modernize its UI in a project for which we integrated Bootstrap. I was very impressed with both AngularJS and Bootstrap and have used them ever since. In 2014, I used Ionic on a project to implement a HTML5 UI in a native iOS application. We used AngularJS, Bootstrap, and Spring Boot in that project and they worked very well for us.

When I heard about JHipster, I was motivated to use it right away. It combined my most-often-used frameworks into an easy-to-use package. For the first several months I knew about JHipster, I used it as a learning tool — generating projects and digging into files to see how it coded features. The JHipster project is a goldmine of information and lessons from several years of developer experience.

I wanted to write this book because I knew all the tools in JHipster really well. I wanted to further the knowledge of this wonderful project. I wanted Java developers to see that they can be hip again by leveraging AngularJS and Bootstrap. I wanted to show them how JavaScript web development isn't scary, it's just another powerful platform that can improve your web-development skills.

The first version of this book was released in October 2015, the second in December 2016, and the third (4.x to match Angular) was released in the fall of 2017 after JHipster migrated to Angular. Version 4.5 was released in April 2018 with an additional chapter on microservices. This version is updated for JHipster version 5. I spent a lot of time learning Angular and TypeScript in 2016-2018 and I'm pleased to bring you this book as an active member of the JHipster Development Team.

What is in an InfoQ mini-book?

InfoQ mini-books are designed to be concise, intending to serve technical architects looking to get a firm conceptual understanding of a new technology or technique in a quick yet in-depth fashion. You can think of these books as covering a topic strategically or essentially. After reading a mini-book, the reader should have a fundamental understanding of a technology, including when and where to apply it, how it relates to other technologies, and an overall feeling that they have assimilated the combined knowledge of other professionals who have already figured out what this technology is about. The reader will then be able to make intelligent decisions about the technology once their projects require it, and can delve into sources of more detailed information (such as larger books or tutorials) at that time.

Who this book is for

This book is aimed specifically at web developers who want a rapid introduction to Angular, Bootstrap, and Spring Boot by learning JHipster. JHipster 5 adds support for React as well, but I won't be covering it in this book since Angular is the default. If I added React coverage, this would no longer be a mini-book. ☺



If you'd like to see a React version of the app developed in this book, check out [Roberto Melfi's 21-points-react project](#).

What you need for this book

To try code samples in this book, you will need a computer running an up-to-date operating system (Windows, Linux, or Mac OS X/macOS). You will need Node.js and Java installed. The book code was tested against Node.js 8 and JDK 8, but newer versions should also work.



It should be possible to run a JHipster app on Java 9+ because it's possible to use Java 10 with Spring Boot 2.0.1. Java 11 support is in Spring Boot 2.1. However, JHipster depends on many plugins that are not yet Java 9+ compatible. The most common error is `java.lang.NoClassDefFoundError: javax/xml/bind/JAXBException`. For this reason, I recommend using Java 8.

Conventions

We use a number of typographical conventions within this book that distinguish between different kinds of information.

Code in the text, including commands, variables, file names, CSS class names, and property names are shown as follows.

Spring Boot uses a `public static void main` entry-point that launches an embedded web server for you.

A block of code is set out as follows. It may be colored, depending on the format in which you're reading this book.

src/app/search/search.component.html

```
<form>
  <input type="search" name="query" [(ngModel)]="query">
  <button type="button" (click)="search()">Search</button>
</form>
```

src/main/java/demo/DemoApplication.java

```
@RestController
class BlogController {
  private final BlogRepository repository;

  // Yay! No annotations needed for constructor injection in Spring 4.3+.
  public BlogController(BlogRepository repository) {
    this.repository = repository;
  }

  @RequestMapping("/blogs")
  Collection<Blog> list() {
    return repository.findAll();
  }
}
```

When we want to draw your attention to certain lines of code, those lines are annotated using numbers accompanied by brief descriptions.

```
export class SearchComponent {
  constructor(private searchService: SearchService) {} ①

  search(): void { ②
    this.searchService.search(this.query).subscribe( ③
      data => { this.searchResults = data; },
      error => console.log(error)
    );
  }
}
```

① To inject `SearchService` into `SearchComponent`, add it as a parameter to the constructors's argument list.

② `search()` is a method that's called from the HTML's `<button>`, wired up using the `(click)` event handler.

③ `this.query` is a variable that's wired to `<input>` using two-way binding with `[(ngModel)]="query"`.



Tips are shown using callouts like this.



Warnings are shown using callouts like this.

Sidebar

Additional information about a certain topic may be displayed in a sidebar like this one.

Finally, this text shows what a quote looks like:

In the end, it's not the years in your life that count. It's the life in your years.

— Abraham Lincoln

Reader feedback

We always welcome feedback from our readers. Let us know what you thought about this book — what you liked or disliked. Reader feedback helps us develop titles that deliver the most value to you.

To send us feedback, email us at feedback@infoq.com, send a tweet to @jhipster_book, or post a question on Stack Overflow using the “jhipster” tag.

If you’re interested in writing a mini-book for InfoQ, see <http://www.infoq.com/minibook-guidelines>.

Introduction

JHipster is one of those open-source projects you stumble upon and immediately think, “Of course!” It combines three very successful frameworks in web development: Bootstrap, Angular, and Spring Boot. Bootstrap was one of the first dominant web-component frameworks. Its largest appeal was that it only required a bit of HTML and it worked! All the efforts we made in the Java community to develop web components were shown a better path by Bootstrap. It leveled the playing field in HTML/CSS development, much like Apple’s Human Interface Guidelines did for iOS apps.

JHipster was started by Julien Dubois in October 2013 (Julien’s first commit was on [October 21, 2013](#)). The first public release (version 0.3.1) was launched December 7, 2013. Since then, the project has had over 172 releases! It is an open-source, Apache 2.0-licensed project on GitHub. It has a core team of 19 developers and over 430 contributors. You can find its homepage at <https://www.jhipster.tech>. Its [GitHub project](#) shows it’s mostly written in JavaScript (33%), Java (27%), and TypeScript (25%). HTML is trailing in the fourth position with 12%.

At its core, JHipster is a [Yeoman](#) generator. Yeoman is a code generator that you run with a `yo` command to generate complete applications or useful pieces of an application. Yeoman generators promote what the Yeoman team calls the “Yeoman workflow”. This is an opinionated client-side stack of tools that can help developers quickly build beautiful web applications. It takes care of providing everything needed to get working without the normal pains associated with a manual setup.

The Yeoman workflow is made up of three types of tools to enhance your productivity and satisfaction when building a web app:

- the scaffolding tool (`yo`),
- the build tool (`npm/Yarn`, `webpack`, etc.), and
- the package manager (`npm/Yarn`).

This book shows you how to build an app with JHipster, and guides you through the plethora of tools, techniques, and options. Furthermore, it explains the UI components and API building blocks so you can understand the underpinnings of a JHipster application.

PART ONE

Building an app with JHipster

When I started writing this book, I had a few different ideas for a sample application. My first idea involved creating a photo gallery to showcase the 1966 VW Bus I've been working on since 2006. The project was recently finished and I wanted a website to show how things have progressed through the years.

I also thought about creating a blog application. As part of [my first presentation on JHipster](#) (at the [Denver Java Users Group](#)), I created a blog application that I live-coded in front of the audience. After that presentation, I spent several hours polishing the application and started [The JHipster Mini-Book site](#) with it.

After thinking about the VW Bus Gallery and developing the blog application, I thought to myself, is this hip enough? Shouldn't a book about becoming what the JHipster homepage calls a "Java Hipster" show how to build a hip application?

I wrote to Julien Dubois, founder of JHipster, and Dennis Sharpe, the technical editor for this book, and asked them what they thought. We went back and forth on a few ideas: a [Gitter](#) clone, a job board for JHipster coders, a shopping-cart app. Then it hit me: there was an idea I'd been wanting to develop for a while.

It's basically an app that you can use to monitor your health. In late September through mid-October 2014, I'd done a sugar detox during which I stopped eating sugar, started exercising regularly, and stopped drinking alcohol. I'd had high blood pressure for over 10 years and was on blood-pressure medication at the time. During the first week of the detox, I ran out of blood-pressure medication. Since a new prescription required a doctor visit, I decided I'd wait until after the detox to get it. After three weeks, not only did I lose 15 pounds, but my blood pressure was at normal levels!

Before I started the detox, I came up with a 21-point system to see how healthy I was being each week. Its rules were simple: you can earn up to three points per day for the following reasons:

1. If you eat healthy, you get a point. Otherwise, zero.
2. If you exercise, you get a point.
3. If you don't drink alcohol, you get a point.

I was surprised to find I got eight points the first week I used this system. During the detox, I got 16 points the first week, 20 the second, and 21 the third. Before the detox, I thought eating healthy meant eating anything except fast food. After the detox, I realized that eating healthy for me meant eating no sugar. I'm also a big lover of craft beer, so I modified the alcohol rule to allow two healthier alcohol drinks (like a greyhound or red wine) per day.

My goal is to earn 15 points per week. I find that if I get more, I'll likely lose weight and have good blood pressure. If I get fewer than 15, I risk getting sick. I've been tracking my health like this since September 2014. I've lost 30 pounds and my blood pressure has returned to and maintained normal levels. I haven't had good blood pressure since my early 20s, so this has been a life changer for me.

I thought writing a "21-Point Health" application would work because tracking your health is always

important. Wearables that can track your health stats might be able to use the APIs or hooks I create to record points for a day. Imagine hooking into [dailymile](#) (where I track my exercise) or [Untappd](#) (where I sometimes list the beers I drink). Or even displaying other activities for a day (e.g. showing your blood-pressure score that you keep on [iOS Health](#)).

I thought my idea would fit nicely with JHipster and Spring Boot from a monitoring standpoint. Spring Boot has lots of health monitors for apps, and now you can use this JHipster app to monitor your health!

Creating the application

I started using the [Installing JHipster](#) instructions. I'm a Java developer, so I already had Java 8 installed, as well as Maven and Git. I installed Node.js from [Nodejs.org](#) and Yarn using [its instructions](#), then ran the following command to install JHipster.

```
npm install -g generator-jhipster
```



If you need to install Java, Maven, or Git, please see [JHipster's local installation documentation](#).

Then I proceeded to build my application. Unlike many application generators in Javaland, Yeoman expects you to be in the directory you want to create your project in, rather than creating the directory for you. So I created a **21-points** directory and typed the following command in it to invoke JHipster.

```
jhipster
```

After running this command, I was prompted to answer questions about how I wanted my application to be generated. You can see the choices I made in the following screenshot.



Figure 1. Generating the application



I tried using "21-points" as the application name, but quickly discovered that this caused issues with the generated TypeScript class names. Starting a class name with a number is illegal, just like it is in Java.

This process generates a `.yo-rc.json` file that captures all of the choices you make. You can use this file in an empty directory to create a project with the same settings.

yo-rc.json

```
{
  "generator-jhipster": {
    "promptValues": {
      "packageName": "org.jhipster.health",
      "nativeLanguage": "en"
    },
    "jhipsterVersion": "5.2.1",
    "applicationType": "monolith",
    "baseName": "TwentyOnePoints",
    "packageName": "org.jhipster.health",
    "packageFolder": "org/jhipster/health",
    "serverPort": "8080",
    "authenticationType": "jwt",
    "cacheProvider": "ehcache",
    "enableHibernateCache": true,
    "websocket": false,
    "databaseType": "sql",
    "devDatabaseType": "h2Disk",
    "prodDatabaseType": "postgresql",
    "searchEngine": "elasticsearch",
    "messageBroker": false,
    "serviceDiscoveryType": false,
    "buildTool": "gradle",
    "enableSwaggerCodegen": false,
    "jwtSecretKey": "43d99bb3ca6ffaa96b33b632e4a82fb45b672d80",
    "clientFramework": "angularX",
    "useSass": true,
    "clientPackageManager": "yarn",
    "testFrameworks": [
      "gatling",
      "protractor"
    ],
    "jhiPrefix": "jhi",
    "enableTranslation": true,
    "nativeLanguage": "en",
    "languages": [
      "en",
      "fr"
    ]
  }
}
```

You can see that I chose H2 with disk-based persistence for development and PostgreSQL for my production database. I did this because using a non-embedded database offers some important

benefits:

- Your data is retained when restarting the application.
- Your application starts a bit faster.
- You can use Liquibase to generate a database changelog.

The [Liquibase](#) homepage describes it as source control for your database. It will help create new fields as you add them to your entities. It will also refactor your database, for example creating tables and dropping columns. It also has the ability to undo changes to your database, either automatically or with custom SQL.

After answering all the questions, JHipster created a whole bunch of files, then ran `yarn install`. To prove everything was good to go, I ran the Java unit tests using `./gradlew test`.

```
BUILD SUCCESSFUL in 1m 7s
10 actionable tasks: 9 executed, 1 up-to-date
```

JHipster v5 will only work with an external Elasticsearch instance. In previous versions, you could use an embedded Elasticsearch instance, but Elasticsearch has removed this ability in recent releases. The easiest way to run a local Elasticsearch instance is to use Docker Compose. I ran the following command to start Elasticsearch as a daemon. Remove the `-d` option if you don't want it to run as a daemon.

```
docker-compose -f src/main/docker/elasticsearch.yml up -d
```



Running Elasticsearch as an embedded instance is the default again in JHipster 5.3.0+ thanks to [this pull-request](#) that integrates Spring Data Jest.

Next, I started the app using `./gradlew` and then ran the UI integration tests with `yarn e2e`. All tests passed with flying colors.

```
$ yarn e2e
yarn run v1.9.4
$ protractor src/test/javascript/protractor.conf.js
(node:30302) [DEP0022] DeprecationWarning: os.tmpDir() is deprecated. Use os.tmpdir()
instead.
[14:45:36] W/configParser - pattern ./e2e/entities/**/*.spec.ts did not match any files.
[14:45:36] I/launcher - Running 1 instances of WebDriver
[14:45:36] I/direct - Using ChromeDriver directly...
Started
.....
11 specs, 0 failures
Finished in 21.309 seconds

[14:45:59] I/launcher - 0 instance(s) of WebDriver still running
[14:45:59] I/launcher - chrome #01 passed
  Done in 23.25s.
```

To prove the `prod` profile worked and I could talk to PostgreSQL, I ran Docker Compose for PostgreSQL.

```
docker-compose -f src/main/docker/postgresql.yml up -d
```

Then I restarted the app with the `prod` profile enabled.

```
$ ./gradlew -Pprod
...
-----
Application 'TwentyOnePoints' is running! Access URLs:
Local:      http://localhost:8080
External:    http://192.168.105.207:8080
Profile(s): [prod]
-----
```

Wahoo — it worked!

Using a local PostgreSQL database

You can also use a local PostgreSQL database. To do this on a Mac, I installed [Postgres.app](#) and tried creating a local PostgreSQL database with settings from [src/main/resources/config/application-prod.yml](#).

```
psql (9.6.10)
Type "help" for help.

template1=# create user TwentyOnePoints with password '21points';
CREATE ROLE
template1=# create database TwentyOnePoints;
CREATE DATABASE
template1=# grant all privileges on database TwentyOnePoints to TwentyOnePoints;
GRANT
template1=#

```

I updated [application-prod.yml](#) to use `21points` for the datasource password. I confirmed I could talk to a PostgreSQL database when running with the `prod` profile. I was greeted with an error saying that things were not set up correctly.

```
$ ./gradlew -Pprod
...
2018-08-27 09:59:32.094 ERROR 5180 --- [           main] com.zaxxer.hikari.pool.HikariPool      :
HikariPool-1 - Exception during pool initialization.

org.postgresql.util.PSQLException: FATAL: role "TwentyOnePoints" does not exist
        at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl.java:2440)
        at org.postgresql.core.v3.QueryExecutorImpl.readStartupMessages(QueryExecutorImpl.java:2559)
        at org.postgresql.core.v3.QueryExecutorImpl.<init>(QueryExecutorImpl.java:133)
```

I quickly realized that PostgreSQL is case insensitive, so even though I typed `"TwentyOnePoints"`, it configured the database name and username as `"twentyonepoints"`. I updated [application-prod.yml](#) with the correct case and tried again. This time it worked!

Adding source control

One of the first things I like to do when creating a new project is to add it to a version-control system (VCS). In this particular case, I chose Git and Bitbucket.

JHipster will initialize Git for your project automatically if you have Git installed. The following commands show how I added a reference to the remote Bitbucket repository, then pushed everything. I created the repository on Bitbucket before executing these commands.

```
$ git remote add origin git@bitbucket.org:mraible/21-points.git
$ git push origin master
Delta compression using up to 8 threads.
Compressing objects: 100% (495/495), done.
Writing objects: 100% (523/523), 616.70 KiB | 8.94 MiB/s, done.
Total 523 (delta 61), reused 0 (delta 0)
remote: Resolving deltas: 100% (61/61), done.
To bitbucket.org:mraible/21-points.git
 * [new branch]      master -> master
```

This is how I created a new application with JHipster and checked it into source control. If you're creating an application following similar steps, I believe there are two common approaches for continuing. The first involves developing the application, then testing and deploying. The second option is to set up continuous integration, deploy, then begin development and testing. In a team development environment, I recommend the second option. However, since you're likely reading this as an individual, I'll follow the first approach and get right to coding. If you're interested in setting up continuous integration with Jenkins, please see [Setting up Continuous Integration on Jenkins 2](#).

Building the UI and business logic

I wanted 21-Points Health to be a bit more hip than a stock JHipster application. Bootstrap was all the rage a few years ago, but now Google's [Material Design](#) is growing in popularity. I searched for "material" in the [JHipster Marketplace](#) and found the [Bootstrap Material Design](#) module. Unfortunately, I soon found out it doesn't support JHipster 4+.

In v4 of this book (and 21-Points Health), I opted to use Bootstrap and its default theme; changing some variables so it looked like Angular Material. Since I got used to it, I decided to keep this same setup for this version. To make the default Bootstrap theme look like Material Design, modify [_bootstrap-variables.scss](#) and replace it with the contents below.

`src/main/webapp/content/scss/_bootstrap-variables.scss`

```
/*
 * Bootstrap overrides https://getbootstrap.com/docs/4.0/getting-started/theming/
 * All values defined in bootstrap source
 * https://github.com/twbs/bootstrap/blob/v4-dev/scss/_variables.scss can be overwritten
 * here
 * Make sure not to add !default to values here
 */

// Colors:
// Grayscale and brand colors for use across Bootstrap.

// Customize colors to match Bootstrap Material Theme
```

```
// https://github.com/FezVrasta/bootstrap-material-
design/blob/master/sass/_variables.scss

$primary: #009688;
$success: #4caf50;
$info: #03a9f4;
$warning: #ff5722;
$danger: #f44336;
$blue: #0275d8;

// Options:
// Quickly modify global styling by enabling or disabling optional features.
$enable-rounded: true;
$enable-shadows: false;
$enable-gradients: false;
$enable-transitions: true;
$enable-hover-media-query: false;
$enable-grid-classes: true;
$enable-print-styles: true;

// Components:
// Define common padding and border radius sizes and more.

$border-radius: 0.15rem;
$border-radius-lg: 0.125rem;
$border-radius-sm: 0.1rem;

// Body:
// Settings for the `<body>` element.

$body-bg: #fff;

// Typography:
// Font, line-height, and color for body text, headings, and more.

$font-size-base: 0.9rem;

$border-radius: 2px;
$border-radius-sm: 1px;

$font-family-sans-serif: 'Roboto', 'Helvetica', 'Arial', sans-serif;
$headers-font-weight: 300;

$link-color: $primary;

$input-focus-border-color: lighten($blue, 25%);
$input-focus-box-shadow: none;
```

Then add the following Sass to the bottom of `global.scss`.

```
/*
=====
custom styles for 21-Points Health
=====
*/
.jh-card {
    border: none !important;
}

.jh-navbar {
    background-color: #009688 !important;
}

blockquote {
    padding: 0.5rem 1rem;
    margin-bottom: 1rem;
    font-size: 1rem !important;
    font-weight: 100;
    border-left: 0.25rem solid #eceeef;
}

a {
    font-weight: normal !important;
}

.truncate {
    width: 180px;
    white-space: nowrap;
    overflow: hidden;
    text-overflow: ellipsis;
    cursor: pointer;

    &.cal-day-notes {
        width: 150px;
    }
}

.footer {
    bottom: 0;
    left: 0;
    color: #666;
    background: #eee;
    border-top: 1px solid silver;
    position: fixed;
    width: 100%;
    padding: 10px;
```

```
padding-bottom: 0;
text-align: center;
z-index: 2;
font-size: 0.9em;

p {
    margin-bottom: 7px;
}
}

.thread-dump-modal-lock {
    max-width: 450px;
    overflow: hidden;
    text-overflow: ellipsis;
    white-space: nowrap;
}

/* Override Bootstrap's default vertical-align: top */
.table {
    th,
    td {
        vertical-align: middle !important;
    }
}
```

How to use Bootstrap Material Design with JHipster 5

If you'd like to use [Bootstrap Material Design](#) with JHipster 5, that's possible too.

Below are the steps necessary to use Bootstrap Material Design and Sass:

1. Install bootstrap-material-design:

```
npm install bootstrap-material-design@4.1.1
```

2. Remove all variables from `src/main/webapp/content/scss/_bootstrap-variables.scss`.
3. Comment out the import for Bootstrap in `src/main/webapp/content/scss/vendor.scss`:

```
// Import Bootstrap source files from node_modules
// @import '~bootstrap/scss/bootstrap';
```

4. Add the following to import bootstrap-material-design in `src/main/webapp/content/scss/vendor.scss`:

```
// Import Bootstrap Material Design
@import url('https://fonts.googleapis.com/css?family=Roboto:300,400,500,700|Material+Icons');
@import '~bootstrap-material-design/scss/bootstrap-material-design';
```

5. Remove the following styles from `global.scss`:

```
/* Error highlight on input fields */
.ng-valid[required],
.ng-valid.required {
    border-left: 5px solid green;
}

.ng-invalid:not(form) {
    border-left: 5px solid red;
}
```

6. Add the following overrides on the dropdown menus so they look good.

```
.dropdown-menu .dropdown-item.active, .dropdown-menu .dropdown-item:active {
  color: #fff !important;
}

.dropdown-menu .dropdown-item {
  display: inline-block !important;
  padding: 0.5rem 1.5rem !important;
  min-height: 2rem !important;
}
```

Below is a screenshot taken after these changes.

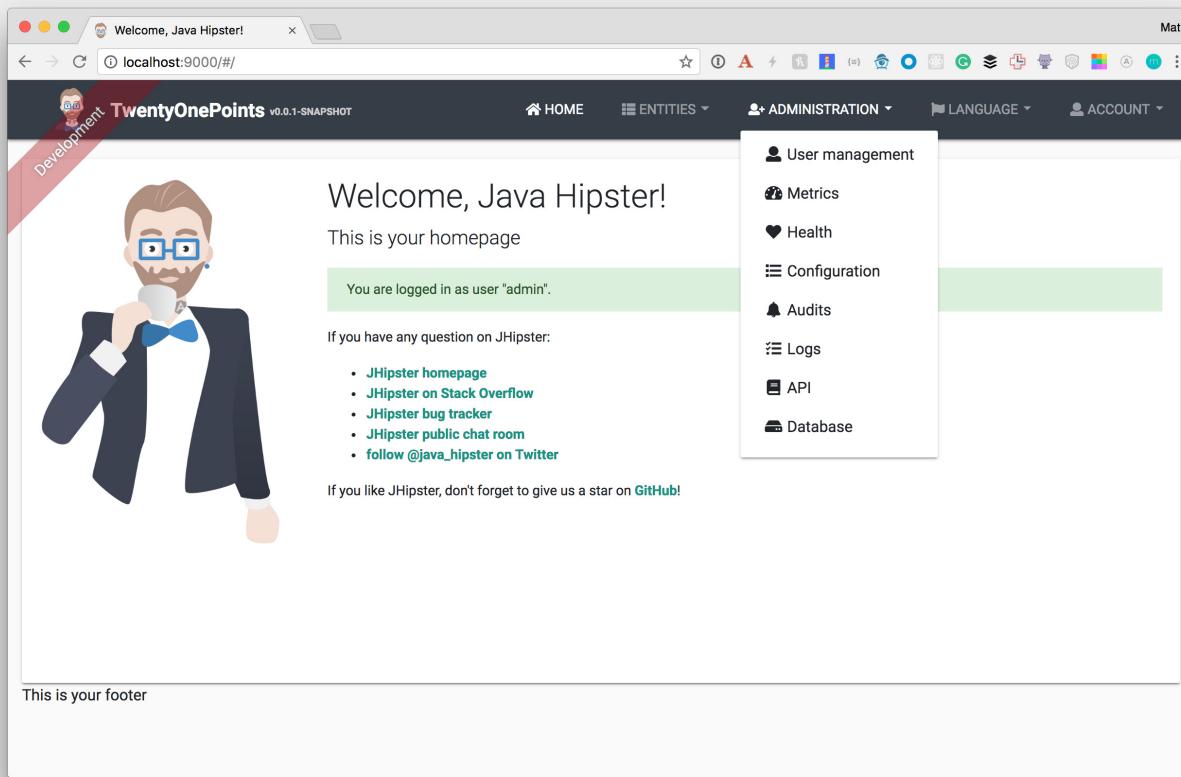


Figure 2. JHipster with Bootstrap Angular Material

At this point, I deployed to Heroku for the first time. This is covered in the [Deploying to Heroku](#) section of this chapter.

Generating entities

For each entity you want to create, you will need:

- a database table;

- a Liquibase change set;
- a JPA entity class;
- a Spring Data `JpaRepository` interface;
- a Spring MVC `RestController` class;
- an Angular router, controller, and service; and
- a HTML page.

In addition, you should have integration tests to verify that everything works and performance tests to verify that it runs fast. In an ideal world, you'd also have unit tests and integration tests for your Angular code.

The good news is JHipster can generate all of this code for you, including integration tests and performance tests. In addition, if you have entities with relationships, it will generate the necessary schema to support them (with foreign keys), and the TypeScript and HTML code to manage them. You can also set up validation to require certain fields as well as control their length.

JHipster supports several methods of code generation. The first uses its [entity sub-generator](#). The entity sub-generator is a command-line tool that prompts you with questions to answer. [JDL-Studio](#) is a browser-based tool for defining your domain model with JHipster Domain Language (JDL). Finally, [JHipster-UML](#) is an option for those that like UML. Supported UML editors include [Modelio](#), [UML Designer](#), [GenMyModel](#) and [Visual Paradigm](#). Because the entity sub-generator is one of the simplest to use, I chose that for this project.



If you want to see how easy it is to use JDL-Studio, please see my [Get Started with JHipster 5 Screencast](#).

At this point, I did some trial-and-error designs with the data model. I generated entities with JHipster, tried the app, and changed to start with a UI-first approach. As a user, I was hoping to easily add daily entries about whether I'd exercised, ate healthy meals, or consumed alcohol. I also wanted to record my weight and blood-pressure metrics when I measured them. When I started using the UI I'd just created, it seemed like it might be able to accomplish these goals, but it also seemed somewhat cumbersome. That's when I decided to create a UI mockup with the main screen and its ancillary screens for data entry. I used [OmniGraffle](#) and a [Bootstrap stencil](#) to create the following UI mockup.

The UI mockup displays five screens:

- 21-Points Health:** Shows a progress bar from 12 to 21 points this week. Includes a "Graph" section.
- Add Weight:** Form fields for Date / Time, Pounds / Kilograms, with Save and Cancel buttons.
- Add Blood Pressure:** Form fields for Date / Time, Systolic, Diastolic, with Save and Cancel buttons.
- Enter Points:** Form fields for Date, Exercise (checkboxes for Exercise, Meals, Alcohol), Notes, and a Graph section. Includes Save and Cancel buttons.
- Settings:** Form fields for Points per Week Goal, Number of points, Weight Units (dropdown with Pounds selected), and a Graph section. Includes Save and Cancel buttons.

Figure 3. UI mockup

After figuring out how I wanted the UI to look, I started to think about the data model. I quickly decided I didn't need to track high-level goals (e.g. lose ten pounds in Q3 2018). I was more concerned with tracking weekly goals and 21-Points Health is all about how many points you get in a week. I created the following diagram as my data model.

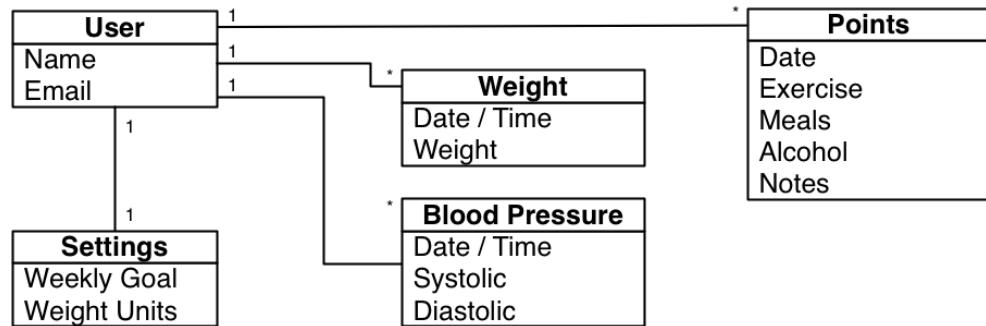


Figure 4. 21-Points Health entity diagram

I ran `jhipster entity points`. I added the appropriate fields and their validation rules, and specified a many-to-one relationship with `user`. Below is the final output from my answers.

```
===== Points =====
```

Fields

```
date (LocalDate) required
exercise (Integer)
meals (Integer)
alcohol (Integer)
notes (String) maxlength='140'
```

Relationships

```
user (User) many-to-one
```

? Do you want to use separate service class for your business logic? No, the REST controller should use the repository directly

? Do you want pagination on your entity? Yes, with pagination links

Everything is configured, generating the entity...

```
create .jhipster/Points.json
create src/main/resources/config/liquibase/changelog/20180828004742_added_entity_Points.xml
create src/main/resources/config/liquibase/changelog/20180828004742_added_entity_constraints_Points.xml
create src/main/java/org/jhipster/health/domain/Points.java
create src/main/java/org/jhipster/health/repository/PointsRepository.java
create src/main/java/org/jhipster/health/web/rest/PointsResource.java
create src/main/java/org/jhipster/health/repository/search/PointsSearchRepository.java
create src/test/java/org/jhipster/health/web/rest/PointsResourceIntTest.java
create src/test/java/org/jhipster/health/repository/search/PointsSearchRepositoryMockConfiguration.java
create src/test/gatling/user-files/simulations/PointsGatlingTest.scala
conflict src/main/resources/config/liquibase/master.xml
? Overwrite src/main/resources/config/liquibase/master.xml? overwrite this and all others
  force src/main/resources/config/liquibase/master.xml
  force src/main/java/org/jhipster/health/config/CacheConfiguration.java
create src/main/webapp/app/entities/points/points.component.html
create src/main/webapp/app/entities/points/points-detail.component.html
create src/main/webapp/app/entities/points/points-update.component.html
create src/main/webapp/app/entities/points/points-delete-dialog.component.html
  force src/main/webapp/app/layouts/navbar/navbar.component.html
create src/main/webapp/i18n/en/points.json
  force src/main/webapp/i18n/en/global.json
create src/main/webapp/i18n/fr/points.json
  force src/main/webapp/i18n/fr/global.json
create src/main/webapp/app/entities/points/index.ts
create src/main/webapp/app/entities/points/points.module.ts
create src/main/webapp/app/entities/points/points.route.ts
create src/main/webapp/app/shared/model/points.model.ts
create src/main/webapp/app/entities/points/points.component.ts
create src/main/webapp/app/entities/points/points-update.component.ts
create src/main/webapp/app/entities/points/points-delete-dialog.component.ts
create src/main/webapp/app/entities/points/points-detail.component.ts
create src/main/webapp/app/entities/points/points.service.ts
create src/test/javascript/spec/app/entities/points/points-detail.component.spec.ts
create src/test/javascript/spec/app/entities/points/points-update.component.spec.ts
create src/test/javascript/spec/app/entities/points/points-delete-dialog.component.spec.ts
create src/test/javascript/spec/app/entities/points/points.component.spec.ts
create src/test/javascript/spec/app/entities/points/points.service.spec.ts
create src/test/javascript/e2e/entities/points/points.page-object.ts
create src/test/javascript/e2e/entities/points/points.spec.ts
  force src/main/webapp/app/entities/entity.module.ts
```

Running 'webpack:build' to update client app

You can see the validation rules for date and notes above, but you don't see how I created the relationship to user. Here are the questions and answers from that section.

```
? Do you want to add a relationship to another entity? Yes
? What is the name of the other entity? user
? What is the name of the relationship? user
? What is the type of the relationship? many-to-one
? When you display this relationship on client-side, which field from 'user' do you want
to use? This field will
be displayed as a String, so it cannot be a Blob login
? Do you want to add any validation rules to this relationship? No
```

I had similar answers for the `Weight` and `BloodPressure` entities. Please refer to the entity diagram for the field names in each entity. For `Preferences`, I created a one-to-one relationship with `User`.

To ensure that people use 21-Points Health effectively, I set the weekly goal to a minimum of 10 points and a max of 21. I also made the `weightUnits` property an enum.

```
===== Preferences =====
```

Fields

```
weeklyGoal (Integer) required min='10' max='21'
```

Generating field #2

```
? Do you want to add a field to your entity? Yes
? What is the name of your field? weightUnits
? What is the type of your field? Enumeration (Java enum type)
? What is the class name of your enumeration? Units
? What are the values of your enumeration (separated by comma, no spaces)? kg,lb
? Do you want to add validation rules to your field? Yes
? Which validation rules do you want to add? Required
```

```
===== Preferences =====
```

Fields

```
weeklyGoal (Integer) required min='10' max='21'
```

```
weightUnits (Units) required
```



After generating the `Weight` and `BloodPressure` entities with a `date` property for the date/time field, I decided that `timestamp` was a better property name. To fix this, I modified the respective JSON files in the `.jhipster` directory and ran `jhipster entity` for each entity again. This seemed easier than refactoring with IntelliJ and hoping it caught all the name instances.

When I ran `./gradlew test`, I was pleased to see that all tests passed.

BUILD SUCCESSFUL in 36s

I checked in six changed files and 130 new files generated by the JHipster before continuing to implement my UI mockups.

Application improvements

To make my new JHipster application into something I could be proud of, I made a number of improvements, described below.



At this point, I set up continuous testing of this project using [Jenkins](#). This is covered in the [Continuous integration and deployment](#) section of this chapter.

Improved HTML layout and I18N messages

Of all the code I write, UI code (HTML, JavaScript, and CSS) is my favorite. I like that you can see changes immediately and make progress quickly - especially when you're using dual monitors with [Browsersync](#). Below is a consolidated list of changes I made to the HTML to make things look better:

- improved layout of tables and buttons,
- improved titles and button labels by editing generated JSON files in `src/main/webapp/i18n/en`,
- formatted dates for local timezone with Angular's DatePipe (for example:
`{{bloodPressure.timestamp | date:'short'}}`),
- defaulted to current date on new entries,
- replaced point metrics with icons on list/detail screens, and
- replaced point metrics with checkboxes on update screen.

The biggest visual improvements are on the list screens. I made the buttons a bit smaller, turned button text into tooltips, and moved add/search buttons to the top right corner. For the points-list screen, I converted the 1 and 0 metric values to icons. Before and after screenshots of the points list illustrate the improved, compact layout.

ID	Date	Did you exercise?	Did you eat well?	Did you drink responsibly?	Notes	User
1001	Aug 19, 2018	1	1	1	Beginning of 14-Day Sugar detox! Took dogs on a hike in backyard. 🚲 ride to CCSP and back. Salmon for dinner.	user
1002	Aug 20, 2018	1	1	1	Rode to Thrive Cherry Creek and back. 36 miles. #pooped	user
1003	Aug 21, 2018	1	1	1	Rode from CCSP to Thrive Ballpark and back. It rained on the way home, but no flooding! Salmon leftovers for lunch, buffalo 🐃 for dinner.	user

21-Points Health | An application developed for a better life and [The JHipster Mini-Book](#) | By Matt Raible

Figure 5. Default Daily Points list

Date	Did you exercise?	Did you eat well?	Did you drink responsibly?	Notes	User
Aug 19, 2018	✓	✓	✓	Beginning of 14-Day Sug...	user
Aug 20, 2018	✓	✓	✓	Rode to Thrive Cherry Cr...	user
Aug 21, 2018	✓	✓	✓	Rode from CCSP to Thriv...	user
Aug 22, 2018	✓	✓	✓	Rode from CCSP to Thriv...	user
Aug 23, 2018	✓	✓	✓	Attempted to smoke ribs. ...	user
Aug 24, 2018	✓	✓	✓	Rode CCSP to Thrive Bal...	user
Aug 25, 2018	✓	✓	✓	Dog walk, golfing, and wa...	user
Aug 26, 2018	✓	✓	✓	🚲 ride this morning. Jac...	user
Aug 27, 2018	✓	✓	✓	Rode to Thrive from CCS...	user

Showing 1 - 9 of 9 items.

21-Points Health | An application developed for a better life and [The JHipster Mini-Book](#) | By Matt Raible

Figure 6. Default Daily Points list after UI improvements

I refactored the HTML at the top of `points.component.html` to put the title, search, and add buttons on the same row. I also removed the button text in favor of using `ng-bootstrap's tooltip directive`. The `jhiTranslate` directive you see in the button tooltips is provided by JHipster's Angular library.

`src/main/webapp/app/entities/points/points.component.html`

```
<div class="row">
    <div class="col-sm-8">
        <h2 id="page-heading" jhiTranslate="twentyOnePointsApp.points.home.title">Points</h2>
    </div>
    <div class="col-sm-4 text-right">
        <button id="jh-create-entity" class="btn btn-primary float-right jh-create-entity create-points"
            [routerLink]=["/points/new"]
            [ngbTooltip]="addTooltip" placement="bottom">
            <fa-icon [icon]="'plus'"></fa-icon>
            <ng-template #addTooltip>
                <span jhiTranslate="twentyOnePointsApp.points.home.createLabel">Add Points</span>
            </ng-template>
        </button>
        <form name="searchForm" class="form-inline">
            <div class="input-group w-100 mr-1">
                <input type="text" class="form-control" [(ngModel)]="currentSearch"
                    id="currentSearch" name="currentSearch"
                    placeholder="{{ 'twentyOnePointsApp.points.home.search' | translate }}">
                <button class="input-group-append btn btn-info" (click)="search(currentSearch)">
                    <fa-icon [icon]="'search'"></fa-icon>
                </button>
                <button class="input-group-append btn btn-danger" (click)="clear()" *ngIf="currentSearch">
                    <fa-icon [icon]="'trash-alt'"></fa-icon>
                </button>
            </div>
        </form>
    </div>
</div>
```

Changing the numbers to icons was pretty easy thanks to Angular's expression language.

src/main/webapp/app/entities/points/points.component.html

```
<td class="text-center">
  <fa-icon [icon]="points.exercise ? 'check' : 'times'" aria-hidden="true"
            class="{{points.exercise ? 'text-success' : 'text-danger'}}"></fa-icon>
</td>
<td class="text-center">
  <fa-icon [icon]="points.meals ? 'check' : 'times'" aria-hidden="true"
            class="{{points.meals ? 'text-success' : 'text-danger'}}"></fa-icon>
</td>
<td class="text-center">
  <fa-icon [icon]="points.alcohol ? 'check' : 'times'" aria-hidden="true"
            class="{{points.alcohol ? 'text-success' : 'text-danger'}}"></fa-icon>
</td>
```

After adding this HTML, I saw an error in my browser's developer console.

```
FontAwesome: Could not find icon with iconName=check and prefix=fas
```

To fix this, I modified `vendor.ts` and added `faCheck` as an imported icon.

src/main/webapp/app/vendor.ts

```
library.add(faTimes);
library.add(faCheck); // add this line
```

Next, I changed the input fields to checkboxes in `points-update.component.html`.

src/main/webapp/app/entities/points/points-update.component.html

```
<div class="form-check">
    <label class="form-check-label" for="field_exercise">
        <input type="checkbox" class="form-check-input" name="exercise" id="field_exercise"
            [(ngModel)]="points.exercise" />
        <span jhiTranslate="twentyOnePointsApp.points.exercise" for="field_exercise">Exercise</span>
    </label>
</div>
<div class="form-check">
    <label class="form-check-label" for="field_meals">
        <input type="checkbox" class="form-check-input" name="meals" id="field_meals"
            [(ngModel)]="points.meals" />
        <span jhiTranslate="twentyOnePointsApp.points.meals">Meals</span>
    </label>
</div>
<div class="form-check">
    <label class="form-check-label" for="field_alcohol">
        <input type="checkbox" class="form-check-input" name="alcohol" id="field_alcohol"
            [(ngModel)]="points.alcohol" />
        <span jhiTranslate="twentyOnePointsApp.points.alcohol" for="field_alcohol">Alcohol</span>
    </label>
</div>
```

In `points-update.component.ts`, I had to modify the `save()` method to convert booleans from each checkbox into integers.

src/main/webapp/app/entities/points/points-update.component.ts

```
save() {
    this.isSaving = true;

    // convert booleans to ints
    this.points.exercise = this.points.exercise ? 1 : 0;
    this.points.meals = this.points.meals ? 1 : 0;
    this.points.alcohol = this.points.alcohol ? 1 : 0;

    if (this.points.id !== undefined) {
        this.subscribeToSaveResponse(this.pointsService.update(this.points));
    } else {
        this.subscribeToSaveResponse(this.pointsService.create(this.points));
    }
}
```

After making these changes, modifying a bit of HTML, and tweaking some i18n messages, the “Add Points” screen is starting to look like the UI mockup I created.

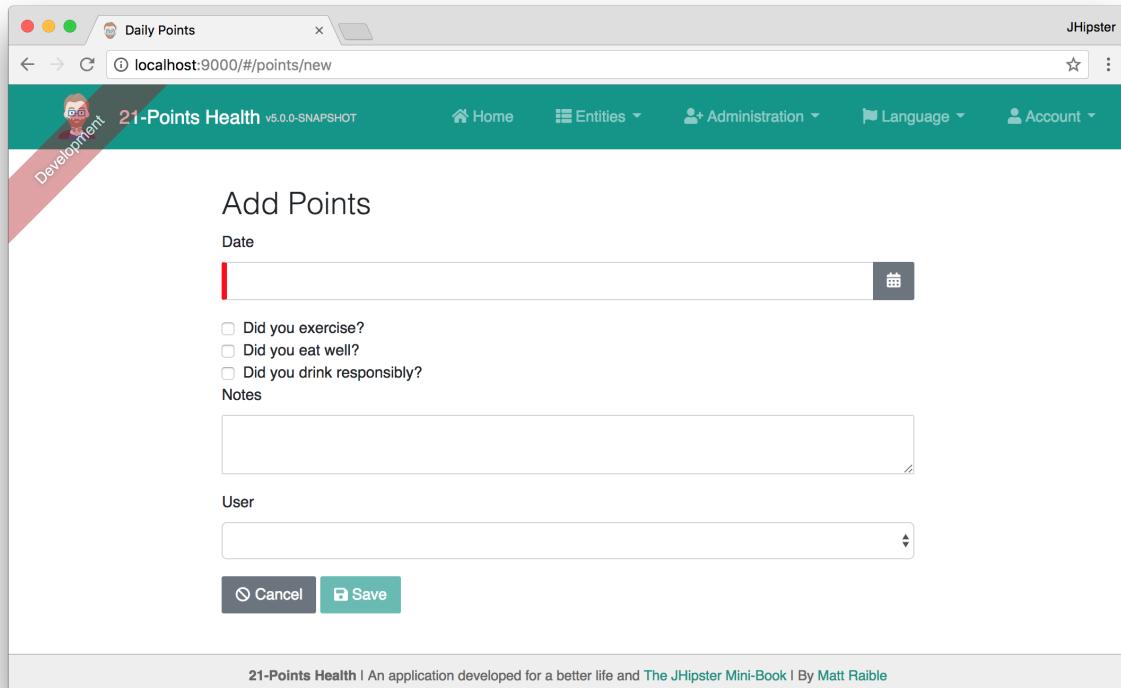


Figure 7. Add Points page

Improving the UI was the most fun, but also the most time consuming as it involved lots of little tweaks to multiple screens. The next task was more straightforward: implementing business logic.

Added logic so non-admin users only see their own data

I wanted to make several improvements to what users could see, based on their roles. A user should be able to see and modify their data, but nobody else's. I also wanted to ensure that an administrator could see and modify everyone's data.

Hide user selection from non-admin users

The default update components for many-to-one relationships allow you to choose the user when you add/edit a record. To make it so only administrators had this ability, I modified the update templates and used the `*jhiHasAnyAuthority` directive. This directive is included with JHipster, in `src/main/webapp/app/shared/auth/has-any-authority.directive.ts`. It allows you to pass in a single role or a list of roles.

src/main/webapp/app/entities/points/points-update.component.html

```
<div class="form-group" *jhiHasAnyAuthority="'ROLE_ADMIN'>
    <label class="form-control-label" jhiTranslate="twentyOnePointsApp.points.user" for="field_user">User</label>
    <select class="form-control" id="field_user" name="user" [(ngModel)]="points.user">
        <option [ngValue]="null"></option>
        <option [ngValue]="userOption.id === points.user?.id ? points.user : userOption"
            *ngFor="let userOption of users; trackBy: trackUserId">{{userOption.login}}</option>
    </select>
</div>
```

Since the dropdown is hidden from non-admins, I had to modify each **Resource** class to default to the current user when creating a new record. Below is a diff that shows the changes that I needed to make to **PointsResource.java**.

src/main/java/org/jhipster/health/web/rest/PointsResource.java

```
+import org.jhipster.health.repository.UserRepository;
+import org.jhipster.health.security.AuthoritiesConstants;
+import org.jhipster.health.security.SecurityUtils;

    private final PointsSearchRepository pointsSearchRepository;

-    public PointsResource(PointsRepository pointsRepository, PointsSearchRepository pointsSearchRepository) {
+    private final UserRepository userRepository;
+
+    public PointsResource(PointsRepository pointsRepository, PointsSearchRepository pointsSearchRepository,
+        UserRepository userRepository) {
        this.pointsRepository = pointsRepository;
        this.pointsSearchRepository = pointsSearchRepository;
+        this.userRepository = userRepository;
    }

    @PostMapping("/points")
    @Timed
    public ResponseEntity<Points> createPoints(@Valid @RequestBody Points points) throws URISyntaxException {
        log.debug("REST request to save Points : {}", points);
        if (points.getId() != null) {
            return ResponseEntity.badRequest().headers(
                HeaderUtil.createFailureAlert(ENTITY_NAME, "idexists",
                    "A new points cannot already have an ID")).body(null);
        }
+        if (!SecurityUtils.isCurrentUserInRole(AuthoritiesConstants.ADMIN)) {
+            log.debug("No user passed in, using current user: {}", SecurityUtils.getCurrentUserLogin());
+            points.setUser(userRepository.findOneByLogin(SecurityUtils.getCurrentUserLogin()).get());
+        }
        Points result = pointsRepository.save(points);
        pointsSearchRepository.save(result);
        return ResponseEntity.created(new URI("/api/points/" + result.getId()))
            .headers(HeaderUtil.createEntityCreationAlert(ENTITY_NAME, result.getId().toString()))
            .body(result);
    }
```

`SecurityUtils` is a class JHipster provides when you create a project. I had to modify `PointsResourceIntTest.java` to be security-aware after making this change.

Spring MVC Test provides a convenient interface called `RequestPostProcessor` that you can use to modify a request. Spring Security provides a number of `RequestPostProcessor` implementations that simplify testing. In order to use Spring Security's `RequestPostProcessor` implementations, you can include them all with the following static import.

```
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*;
```

I then modified `PointsResourceIntTest.java`, creating a new `MockMvc` instance that was security-aware and specified `with(user("user"))` to populate Spring Security's `SecurityContext` with an authenticated user.

src/test/java/org/jhipster/health/web/rest/PointsResourceIntTest.java

```
+import org.jhipster.health.domain.User;
+import org.springframework.web.context.WebApplicationContext;
+import java.time.DayOfWeek;
+import java.time.format.DateTimeFormatter;
+import java.time.temporal.ChronoField;
+import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.user;
+import static org.springframework.security.test.web.servlet.setup.SecurityMockMvcConfigurers.springSecurity;

public class PointsResourceIntTest {
    ...
    @Autowired
    private PointsSearchRepository pointsSearchRepository;

+    @Autowired
+    private UserRepository userRepository;

    ...

+    @Autowired
+    private WebApplicationContext context;
+
+    private MockMvc restPointsMockMvc;

    private Points points;

    @Before
    public void setup() {
        MockitoAnnotations.initMocks(this);
-        PointsResource pointsResource = new PointsResource(pointsRepository, pointsSearchRepository);
+        PointsResource pointsResource = new PointsResource(pointsRepository, pointsSearchRepository, userRepository);
        this.restPointsMockMvc = MockMvcBuilders.standaloneSetup(pointsResource)
            .setCustomArgumentResolvers(pageableArgumentResolver)
            .setControllerAdvice(exceptionTranslator)
            .setMessageConverters(jacksonMessageConverter).build();
    }

    ...

    public void createPoints() throws Exception {
        int databaseSizeBeforeCreate = pointsRepository.findAll().size();

+        // Create security-aware mockMvc
+        restPointsMockMvc = MockMvcBuilders
+            .webAppContextSetup(context)
+            .apply(springSecurity())
+            .build();
+
+        // Create the Points
+        restPointsMockMvc.perform(post("/api/points")
+            .with(user("user"))
+            .contentType(TestUtil.APPLICATION_JSON_UTF8)
+            .content(TestUtil.convertObjectToJsonBytes(points)))
+            .andExpect(status().isCreated());
+
    }
}
```

List screen should show only user's data

The next business-logic improvement I wanted was to modify list screens so they'd only show records for current user. Admin users should see all users' data. To facilitate this feature, I modified `PointsResource#getAll` to have a switch based on the user's role. Rather than showing you the diff of method, here's the whole thing.

src/main/java/org/jhipster/health/web/rest/PointsResource.java

```
public ResponseEntity<List<Points>> getAllPoints(@ApiParam Pageable pageable) {
    log.debug("REST request to get a page of Points");
    Page<Points> page;
    if (SecurityUtils.isCurrentUserInRole(AuthoritiesConstants.ADMIN)) {
        page = pointsRepository.findAllByOrderDateDesc(pageable);
    } else {
        page = pointsRepository.findByUserIsCurrentUser(pageable);
    }
    HttpHeaders headers = PaginationUtil.generatePaginationHttpHeaders(page, "/api/points");
    return new ResponseEntity<>(page.getContent(), headers, HttpStatus.OK);
}
```

The `PointsRepository#findByUserIsCurrentUser()` method that JHipster generated contains a custom query that uses Spring Expression Language to grab the user's information from Spring Security. I changed it from returning `List<Points>` to returning `Page<Points>`.

src/main/java/org/jhipster/health/repository/PointsRepository.java

```
@Query("select points from Points points where points.user.login = ?#{principal.username}")
Page<Points> findByUserIsCurrentUser(Pageable pageable);
```

Ordering by date

Later on, I changed the above query to order by date, so the first records in the list would be the most recent.

src/main/java/org/jhipster/health/repository/PointsRepository.java

```
@Query("select points from Points points where points.user.login = ?#{principal.username} order by points.date desc")
```

In addition, I changed the call to `pointsRepository.findAll` to `pointsRepository.findAllByOrderByDateDesc` so the admin user's query would order by date. The query for this is generated dynamically by Spring Data, simply by adding the method to your repository.

```
Page<Points> findAllByOrderByDateDesc(Pageable pageable);
```

To make tests pass, I had to update `PointsResourceIntTest#getAllPoints` to use Spring Security Test's `user` post processor.

src/test/java/org/jhipster/health/web/rest/PointsResourceIntTest.java

```
@Test
@Transactional
public void getAllPoints() throws Exception {
    // Initialize the database
    pointsRepository.saveAndFlush(points);

    // Create security-aware mockMvc
    restPointsMockMvc = MockMvcBuilders
        .webAppContextSetup(context)
        .apply(springSecurity())
        .build();

    // Get all the points
    restPointsMockMvc.perform(get("/api/points?sort=id,desc"))
        .andExpect(status().isOk());
}
```

Implementing the UI mockup

Making the homepage into something resembling my UI mockup required several steps:

1. Add buttons to facilitate adding new data from the homepage.
2. Add an API to get points achieved during the current week.
3. Add an API to get blood-pressure readings for the last 30 days.
4. Add an API to get body weights for the last 30 days.
5. Add charts to display points per week and blood pressure/weight for last 30 days.

I started by reusing the update components for entering data that JHipster had created for me. I navigated to the components using Angular's `routerLink` syntax, copied from each entity's main list page. For example, below is the code for the "Add Points" button.

```
<a [routerLink]="/points/new" class="btn btn-primary m-0 mb-1 text-white">Add Points</a>
```

Then I had to modify `home.component.ts` to listen for the events these components fire when they modify an entity.

src/main/webapp/app/home/home.component.ts

```
import { JhiEventManager } from 'ng-jhipster';
import { Component, OnDestroy, OnInit } from '@angular/core';
import { Subscription } from 'rxjs';

...
export class HomeComponent implements OnInit, OnDestroy {
    ...
    eventSubscriber: Subscription;

    constructor(..., private eventManager: EventManager) {
    }

    ngOnDestroy() {
        this.eventManager.destroy(this.eventSubscriber);
    }

    registerAuthenticationSuccess() {
        this.eventManager.subscribe('authenticationSuccess', () => {
            this.principal.identity().then((account) => {
                this.account = account;
                this.getUserData();
            });
        });
        this.eventSubscriber = this.eventManager
            .subscribe('pointsListModification', () => this.getUserData());
        this.eventSubscriber = this.eventManager
            .subscribe('bloodPressureListModification', () => this.getUserData());
        this.eventSubscriber = this.eventManager
            .subscribe('weightListModification', () => this.getUserData());
    }
    ...
}
```

Points this week

To get points achieved in the current week, I started by adding a unit test to [PointsResourceIntTest.java](#) that would allow me to prove my API was working.

src/test/java/org/jhipster/health/web/rest/PointsResourceIntTest.java

```

private void createPointsByWeek(LocalDate thisMonday, LocalDate lastMonday) {
    User user = userRepository.findOneByLogin("user").get();
    // Create points in two separate weeks
    points = new Points(thisMonday.plusDays(2), 1, 1, 1, user); ①
    pointsRepository.saveAndFlush(points);

    points = new Points(thisMonday.plusDays(3), 1, 1, 0, user);
    pointsRepository.saveAndFlush(points);

    points = new Points(lastMonday.plusDays(3), 0, 0, 1, user);
    pointsRepository.saveAndFlush(points);

    points = new Points(lastMonday.plusDays(4), 1, 1, 0, user);
    pointsRepository.saveAndFlush(points);
}

@Test
@Transactional
public void getPointsThisWeek() throws Exception {
    LocalDate today = LocalDate.now();
    LocalDate thisMonday = today.with(DayOfWeek.MONDAY);
    LocalDate lastMonday = thisMonday.minusWeeks(1);
    createPointsByWeek(thisMonday, lastMonday);

    // create security-aware mockMvc
    restPointsMockMvc = MockMvcBuilders
        .webAppContextSetup(context)
        .apply(springSecurity())
        .build();

    // Get all the points
    restPointsMockMvc.perform(get("/api/points")
        .with(user("user").roles("USER")))
        .andExpect(status().isOk())
        .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.points", hasSize(4)));

    // Get the points for this week only
    restPointsMockMvc.perform(get("/api/points-this-week")
        .with(user("user").roles("USER")))
        .andExpect(status().isOk())
        .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.week").value(thisMonday.toString()))
        .andExpect(jsonPath("$.points").value(5));
}

```

- ① To simplify testing, I added a new constructor to `Points.java` that contained the arguments I wanted to set. I continued this pattern for most tests I created.

Of course, this test failed when I first ran it since `/api/points-this-week` didn't exist in `PointsResource.java`. You might notice the points-this-week API expects two return values: a date in the `week` field and the number of points in the `points` field. I created `PointsPerWeek.java` in my project's `rest.vm` package to hold this information.

src/main/java/org/jhipster/health/web/rest/vm/PointsPerWeek.java

```
package org.jhipster.health.web.rest.vm;

import java.time.LocalDate;

public class PointsPerWeek {
    private LocalDate week;
    private Integer points;

    public PointsPerWeek(LocalDate week, Integer points) {
        this.week = week;
        this.points = points;
    }

    public Integer getPoints() {
        return points;
    }

    public void setPoints(Integer points) {
        this.points = points;
    }

    public LocalDate getWeek() {
        return week;
    }

    public void setWeek(LocalDate week) {
        this.week = week;
    }

    @Override
    public String toString() {
        return "PointsThisWeek{" +
            "points=" + points +
            ", week=" + week +
            '}';
    }
}
```

Spring Data JPA made it easy to find all point entries in a particular week. I added a new method to my **PointsRepository.java** that allowed me to query between two dates.

src/main/java/org/jhipster/health/repository/PointsRepository.java

```
List<Points> findAllByDateBetweenAndUserLogin(LocalDate firstDate, LocalDate secondDate,
String login);
```

From there, it was just a matter of calculating the beginning and end of the current week and processing the data in [PointsResource.java](#).

src/main/java/org/jhipster/health/web/rest/PointsResource.java

```
/**
 * GET /points : get all the points for the current week.
 */
@GetMapping("/points-this-week")
@Timed
public ResponseEntity<PointsPerWeek> getPointsThisWeek(
    @RequestParam(value="tz", required=false) String timezone) {

    // Get current date (with timezone if passed in)
    LocalDate now = LocalDate.now();
    if (timezone != null) {
        now = LocalDate.now(ZoneId.of(timezone));
    }

    // Get first day of week
    LocalDate startOfWeek = now.with(DayOfWeek.MONDAY);
    // Get last day of week
    LocalDate endOfWeek = now.with(DayOfWeek.SUNDAY);
    log.debug("Looking for points between: {} and {}", startOfWeek, endOfWeek);

    List<Points> points =
        pointsRepository.findAllByDateBetweenAndUserLogin(
            startOfWeek, endOfWeek, SecurityUtils.getCurrentUserLogin());
    return calculatePoints(startOfWeek, points);
}

private ResponseEntity<PointsPerWeek> calculatePoints(LocalDate startOfWeek,
                                                       List<Points> points) {
    Integer numPoints = points.stream()
        .mapToInt(p -> p.getExercise() + p.getMeals() + p.getAlcohol())
        .sum();

    PointsPerWeek count = new PointsPerWeek(startOfWeek, numPoints);
    return new ResponseEntity<>(count, HttpStatus.OK);
}
```

To support this new method on the client, I added a new method to `PointsService` in `src/main/webapp/app/entities/points/points.service.ts`.

`src/main/webapp/app/entities/points/points.service.ts`

```
thisWeek(): Observable<EntityResponseType> {
    const tz = Intl.DateTimeFormat().resolvedOptions().timeZone;
    return this.http
        .get(`api/points-this-week?tz=${tz}`, { observe: 'response' })
        .pipe(map((res: EntityResponseType) => this.convertDateFromServer(res)));
}
```

Then I added the service as a dependency to `home.component.ts` and calculated the data I wanted to display.

src/main/webapp/app/home/home.component.ts

```
import { Account, LoginModalService, Principal } from 'app/core';
import { PreferencesService } from 'app/entities/preferences';
import { BloodPressureService } from 'app/entities/blood-pressure';
import { WeightService } from 'app/entities/weight';
import { D3ChartService } from './d3-chart.service';
import { Preferences } from 'app/shared/model/preferences.model';

...
export class HomeComponent implements OnInit, OnDestroy {
    account: Account;
    modalRef: NgbModalRef;
    pointsThisWeek: any = {};
    pointsPercentage: number;

    constructor(private principal: Principal,
                private loginModalService: LoginModalService,
                private eventManager: EventManager,
                private pointsService: PointsService) {
    }

    getUserData() {
        // Get points for the current week
        this.pointsService.thisWeek().subscribe((points: any) => {
            points = points.body;
            this.pointsThisWeek = points;
            this.pointsPercentage = (points.points / 21) * 100;
        });
    }
    ...
}
```

I added a progress bar to `home.component.html` to show points-this-week progress.

src/main/webapp/app/home/home.component.html

```
<div class="row">
  <div class="col-md-11">
    <ngb-progressbar max="21" [value]="pointsThisWeek.points"
      [hidden]="!pointsThisWeek.points" [striped]="true">
      <span *ngIf="pointsThisWeek.points">
        {{pointsThisWeek.points}} / Goal: 10
      </span>
    </ngb-progressbar>
    <ngb-alert [dismissible]="false" [hidden]="pointsThisWeek.points">
      <span jhiTranslate="home.points.getMoving">
        No points yet this week, better get moving!
      </span>
    </ngb-alert>
  </div>
</div>
```

Below is a screenshot of what this progress bar looked like after restarting the server and entering some data for the current user.

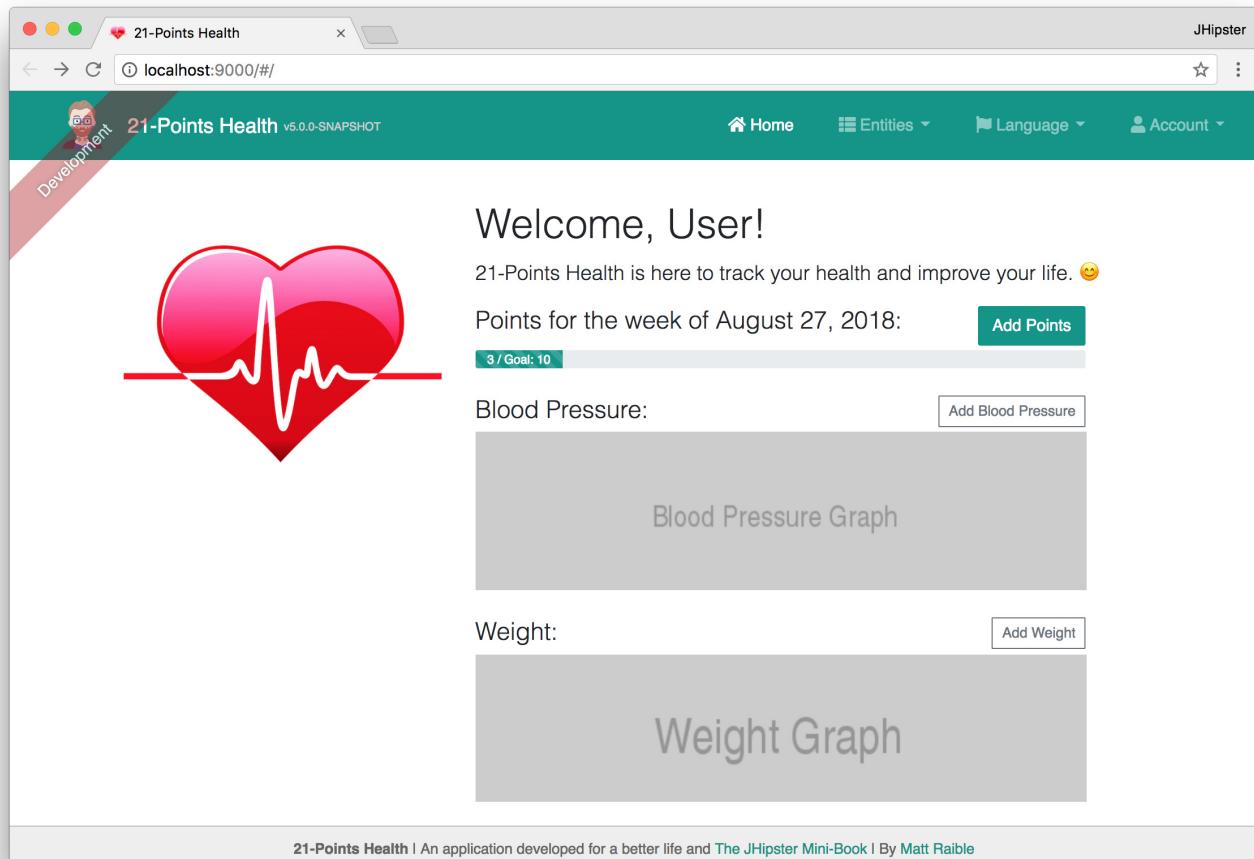


Figure 8. Progress bar for points this week

You might notice the goal is hardcoded to 10 in the progress bar's HTML. To fix this, I needed to add the ability to fetch the user's preferences. To make it easier to access a user's preferences, I modified `PreferencesRepository.java` and added a method to retrieve a user's preferences.

`src/main/java/org/jhipster/health/repository/PreferencesRepository.java`

```
public interface PreferencesRepository extends JpaRepository<Preferences, Long> {
    Optional<Preferences> findOneByUserLogin(String login);
}
```

I created a new method in `PreferencesResource.java` to return the user's preferences (or a default weekly goal of 10 points if no preferences are defined).

`src/main/java/org/jhipster/health/web/rest/PreferencesResource.java`

```
/**
 * GET /my-preferences -> get the current user's preferences.
 */
@GetMapping("/my-preferences")
@Timed
public ResponseEntity<Preferences> getUserPreferences() {
    String username = SecurityUtils.getCurrentUserLogin().get();
    log.debug("REST request to get Preferences : {}", username);
    Optional<Preferences> preferences =
        preferencesRepository.findOneByUserLogin(username);

    if (preferences.isPresent()) {
        return new ResponseEntity<>(preferences.get(), HttpStatus.OK);
    } else {
        Preferences defaultPreferences = new Preferences();
        defaultPreferences.setWeeklyGoal(10); // default
        return new ResponseEntity<>(defaultPreferences, HttpStatus.OK);
    }
}
```

To facilitate calling this endpoint, I added a new `user` method to the `PreferencesService` in the client.

`src/main/webapp/app/entities/preferences/preferences.service.ts`

```
user(): Observable<EntityResponseType> {
    return this.http.get<IPreferences>('api/my-preferences', { observe: 'response' });
}
```

In `home.component.ts`, I added the `PreferencesService` as a dependency and set the preferences in a local `preferences` variable so the HTML template could read it. I also added a listener for `Preference` updates

and logic to calculate the background color of the progress bar.

src/main/webapp/app/home/home.component.ts

```
export class HomeComponent implements OnInit, OnDestroy {
    ...
    preferences: Preferences;

    constructor(
        ...
        private preferencesService: PreferencesService,
        private pointsService: PointsService) {
    }

    registerAuthenticationSuccess() {
        ...
        this.eventSubscriber = this.eventManager.subscribe(
            'preferencesListModification', () => this.getUserData());
    }

    getUserData() {
        // Get preferences
        this.preferencesService.user().subscribe((preferences: any) => {
            this.preferences = preferences.body;

            // Get points for the current week
            this.pointsService.thisWeek().subscribe((points: any) => {
                points = points.body;
                this.pointsThisWeek = points;
                this.pointsPercentage =
                    (points.points / this.preferences.weeklyGoal) * 100;

                // calculate success, warning, or danger
                if (points.points >= preferences.weeklyGoal) {
                    this.pointsThisWeek.progress = 'success';
                } else if (points.points < 10) {
                    this.pointsThisWeek.progress = 'danger';
                } else if (points.points > 10 &&
                    points.points < this.preferences.weeklyGoal) {
                    this.pointsThisWeek.progress = 'warning';
                }
            });
            ...
        });
    }
    ...
}
```

Now that a user's preferences were available, I modified `home.component.html` to display the user's weekly goal, as well as to color the progress bar appropriately with a `[type]` attribute.

`src/main/webapp/app/home/home.component.html`

```
<ngb-progressbar max="21" [value]="pointsThisWeek.points"
                 [type]="pointsThisWeek.progress" [striped]="true"
                 [hidden]="!pointsThisWeek.points">
  <span *ngIf="pointsThisWeek.points">
    {{pointsThisWeek.points}} / Goal: {{preferences.weeklyGoal}}
  </span>a
</ngb-progressbar>
<ngb-alert [dismissible]="false" [hidden]="pointsThisWeek.points">
  <span jhiTranslate="home.points.getMoving">
    No points yet this week, better get moving!
  </span>
</ngb-alert>
```

To finish things off, I added a link to a component where users could edit their preferences.

`src/main/webapp/app/home/home.component.html`

```
<a [routerLink]=["/preferences", preferences.id, 'edit']"
   class="float-right" jhiTranslate="home.link.preferences">Edit Preferences</a>
```

Blood pressure and weight for the last 30 days

To populate the two remaining charts on the homepage, I needed to fetch the user's blood-pressure readings and weights for the last 30 days. I added a method to `BloodPressureResourceIntTest.java` to set up my expectations.

`src/test/java/org/jhipster/health/web/rest/BloodPressureResourceIntTest.java`

```
private void createBloodPressureByMonth(ZonedDateTime firstDate,
                                         ZonedDateTime firstDayOfLastMonth) {
  User user = userRepository.findOneByLogin("user").get();

  bloodPressure = new BloodPressure(firstDate, 120, 80, user);
  bloodPressureRepository.saveAndFlush(bloodPressure);
  bloodPressure = new BloodPressure(firstDate.plusDays(10), 125, 75, user);
  bloodPressureRepository.saveAndFlush(bloodPressure);
  bloodPressure = new BloodPressure(firstDate.plusDays(20), 100, 69, user);
  bloodPressureRepository.saveAndFlush(bloodPressure);

  // last month
  bloodPressure = new BloodPressure(firstDayOfLastMonth, 130, 90, user);
  bloodPressureRepository.saveAndFlush(bloodPressure);
```

```

bloodPressure = new BloodPressure(firstDayOfLastMonth.plusDays(11), 135, 85, user);
bloodPressureRepository.saveAndFlush(bloodPressure);
bloodPressure = new BloodPressure(firstDayOfLastMonth.plusDays(23), 130, 75, user);
bloodPressureRepository.saveAndFlush(bloodPressure);
}

@Test
@Transactional
public void getBloodPressureForLast30Days() throws Exception {
    ZonedDateTime now = ZonedDateTime.now();
    ZonedDateTime twentyNineDaysAgo = now.minusDays(29);
    ZonedDateTime firstDayOfLastMonth = now.withDayOfMonth(1).minusMonths(1);
    createBloodPressureByMonth(twentyNineDaysAgo, firstDayOfLastMonth);

    // Create security-aware mockMvc
    restBloodPressureMockMvc = MockMvcBuilders
        .webAppContextSetup(context)
        .apply(springSecurity())
        .build();

    // Get all the blood pressure readings
    restBloodPressureMockMvc.perform(get("/api/blood-pressures")
        .with(user("user").roles("USER")))
        .andExpect(status().isOk())
        .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.size()", hasSize(6)));

    // Get the blood pressure readings for the last 30 days
    restBloodPressureMockMvc.perform(get("/api/bp-by-days/{days}", 30)
        .with(user("user").roles("USER")))
        .andDo(print())
        .andExpect(status().isOk())
        .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.period").value("Last 30 Days"))
        .andExpect(jsonPath("$.readings[*].systolic").value(hasItem(120)))
        .andExpect(jsonPath("$.readings[*].diastolic").value(hasItem(69)));
}

```

I created a `BloodPressureByPeriod.java` class to return the results from the API.

src/main/java/org/jhipster/health/web/rest/vm/BloodPressureByPeriod.java

```
public class BloodPressureByPeriod {  
    private String period;  
    private List<BloodPressure> readings;  
  
    public BloodPressureByPeriod(String period, List<BloodPressure> readings) {  
        this.period = period;  
        this.readings = readings;  
    }  
    ...  
}
```

Using similar logic that I used for points-this-week, I created a new method in `BloodPressureRepository.java` that allowed me to query between two different dates. I also added “OrderBy” logic so the records would be sorted by date entered.

src/main/java/org/jhipster/health/repository/BloodPressureRepository.java

```
List<BloodPressure> findAllByTimestampBetweenOrderByTimestampDesc(  
    ZonedDateTime firstDate, ZonedDateTime secondDate);
```

Next, I created a new method in `BloodPressureResource.java` that calculated the first and last days of the current month, executed the query for the current user, and constructed the data to return.

src/main/java/org/jhipster/health/web/rest/BloodPressureResource.java

```
/**  
 * GET /bp-by-days : get all the blood pressure readings by last x days.  
 */  
  
@RequestMapping(value = "/bp-by-days/{days}")  
@Timed  
public ResponseEntity<BloodPressureByPeriod> getByDays(@PathVariable int days) {  
    ZonedDateTime rightNow = ZonedDateTime.now(ZoneOffset.UTC);  
    ZonedDateTime daysAgo = rightNow.minusDays(days);  
  
    List<BloodPressure> readings =  
        bloodPressureRepository.findAllByTimestampBetweenOrderByTimestampDesc(daysAgo, rightNow);  
    BloodPressureByPeriod response =  
        new BloodPressureByPeriod("Last " + days + " Days", filterByUser(readings));  
    return new ResponseEntity<>(response, HttpStatus.OK);  
}  
  
private List<BloodPressure> filterByUser(List<BloodPressure> readings) {  
    Stream<BloodPressure> userReadings = readings.stream()  
        .filter(bp -> bp.getUser().getLogin().equals(SecurityUtils.getCurrentLogin().get()));  
    return userReadings.collect(Collectors.toList());  
}
```

Filtering by method

I later learned how to do the filtering in the database by adding the following method to `BloodPressureRepository.java`:

`src/main/java/org/jhipster/health/repository/BloodPressureRepository.java`

```
List<BloodPressure> findAllByTimestampBetweenAndUserLoginOrderByTimestampDesc(
    ZonedDateTime firstDate, ZonedDateTime secondDate, String login);
```

Then I was able to remove the `filterByUser` method and change `BloodPressureResource#getByDays` to be:

`src/main/java/org/jhipster/health/web/rest/BloodPressureResource.java`

```
public ResponseEntity<BloodPressureByPeriod> getByDays(@PathVariable int days) {
    ZonedDateTime rightNow = ZonedDateTime.now();
    ZonedDateTime daysAgo = rightNow.minusDays(days);

    List<BloodPressure> readings =
        bloodPressureRepository.findAllByTimestampBetweenAndUserLoginOrderByTimestampDesc(
            daysAgo, rightNow, SecurityUtils.getCurrentUserLogin().get());
    BloodPressureByPeriod response =
        new BloodPressureByPeriod("Last " + days + " Days", readings);
    return new ResponseEntity<>(response, HttpStatus.OK);
}
```

I added a new method to support this API in `blood-pressure.service.ts`.

`src/main/webapp/app/entities/blood-pressure/blood-pressure.service.ts`

```
last30Days(): Observable<EntityResponseType> {
    return this.http
        .get('api/bp-by-days/30', { observe: 'response' })
        .pipe(map((res: EntityResponseType) => this.convertDateFromServer(res)));
}
```

While gathering this data seemed easy enough, the hard part was figuring out what charting library to use to display it.

Charts of the last 30 days

Based on my experience writing the first two versions of this book, I looked for an Angular library that integrated with `D3.js` and found `ng2-nvd3`. To install ng2-nvd3, I used Yarn's `add` command.

```
yarn add ng2-nvd3
```

Then I updated `home.module.ts` to import the `NvD3Module`, as well as other imports I found necessary.

`src/main/webapp/app/home/home.module.ts`

```
import { NvD3Module } from 'ng2-nvd3';
import 'd3';
import 'nvd3';

@NgModule({
  imports: [
    TwentyOnePointsSharedModule,
    NvD3Module,
    ...
  ],
  ...
})
export class TwentyOnePointsHomeModule {}
```

I modified `home.component.ts` to have the `BloodPressureService` as a dependency and went to work building the data so D3 could render it. I found that charts required a bit of JSON to configure them, so I created a service to contain this configuration.

`src/main/webapp/app/home/d3-chart.service.ts`

```
declare const d3, nv: any;

/**
 * ChartService to define the chart config for D3
 */
export class D3ChartService {

  static getChartConfig() {
    const today = new Date();
    const priorDate = new Date(). setDate(today.getDate() - 30);
    return {
      chart: {
        type: 'lineChart',
        height: 200,
        margin: {
          top: 20,
          right: 20,
          bottom: 40,
          left: 55
        }
      }
    }
  }
}
```

```
        },
        x(d) {
            return d.x;
        },
        y(d) {
            return d.y;
        },
        useInteractiveGuideline: true,
        dispatch: {},
        xAxis: {
            axisLabel: 'Dates',
            showMaxMin: false,
            tickFormat(d) {
                return d3.time.format('%b %d')(new Date(d));
            }
        },
        xDomain: [priorDate, today],
        yAxis: {
            axisLabel: '',
            axisLabelDistance: 30
        },
        transitionDuration: 250
    },
    title: {
        enable: true
    }
);
}
}
```

In `home.component.ts`, I grabbed the blood-pressure readings from the API and morphed them into data that D3 could understand.

src/main/webapp/app/home/home.component.ts

```
// Get blood pressure readings for the last 30 days
this.bloodPressureService.last30Days().subscribe((bpReadings: any) => {
    bpReadings = bpReadings.body;
    this.bpReadings = bpReadings;
    this.bpOptions = {... D3ChartService.getChartConfig() };
    if (bpReadings.readings.length) {
        this.bpOptions.title.text = bpReadings.period;
        this.bpOptions.chart.yAxis.axisLabel = 'Blood Pressure';
        let systolics, diastolics, upperValues, lowerValues;
        systolics = [];
        diastolics = [];
        upperValues = [];
        lowerValues = [];
        bpReadings.readings.forEach((item) => {
            systolics.push({
                x: new Date(item.timestamp),
                y: item.systolic
            });
            diastolics.push({
                x: new Date(item.timestamp),
                y: item.diastolic
            });
            upperValues.push(item.systolic);
            lowerValues.push(item.diastolic);
        });
        this.bpData = [
            {
                values: systolics,
                key: 'Systolic',
                color: '#673ab7'
            },
            {
                values: diastolics,
                key: 'Diastolic',
                color: '#03a9f4'
            }
        ];
        // set y scale to be 10 more than max and min
        this.bpOptions.chart.yDomain =
            [Math.min.apply(Math, lowerValues) - 10,
             Math.max.apply(Math, upperValues) + 10];
    } else {
        this.bpReadings.readings = [];
    }
});
```

Finally, I used the “nvd3” directive in `home.component.html` to read `bpOptions` and `bpData`, then display a

chart.

src/main/webapp/app/home/home.component.html

```
<div class="row mt-1">
  <div class="col-md-11 col-xs-12">
    <span *ngIf="bpReadings.readings && bpReadings.readings.length">
      <nvd3 [options]="bpOptions" [data]="bpData"
        class="with-3d-shadow with-transitions"></nvd3>
    </span>
    <ngb-alert [dismissible]="false"
      [hidden]="bpReadings.readings && bpReadings.readings.length">
      <span jhiTranslate="home.bloodPressure.noReadings">
        No blood pressure readings found.
      </span>
    </ngb-alert>
  </div>
</div>
```

After entering some test data, I was quite pleased with the results.

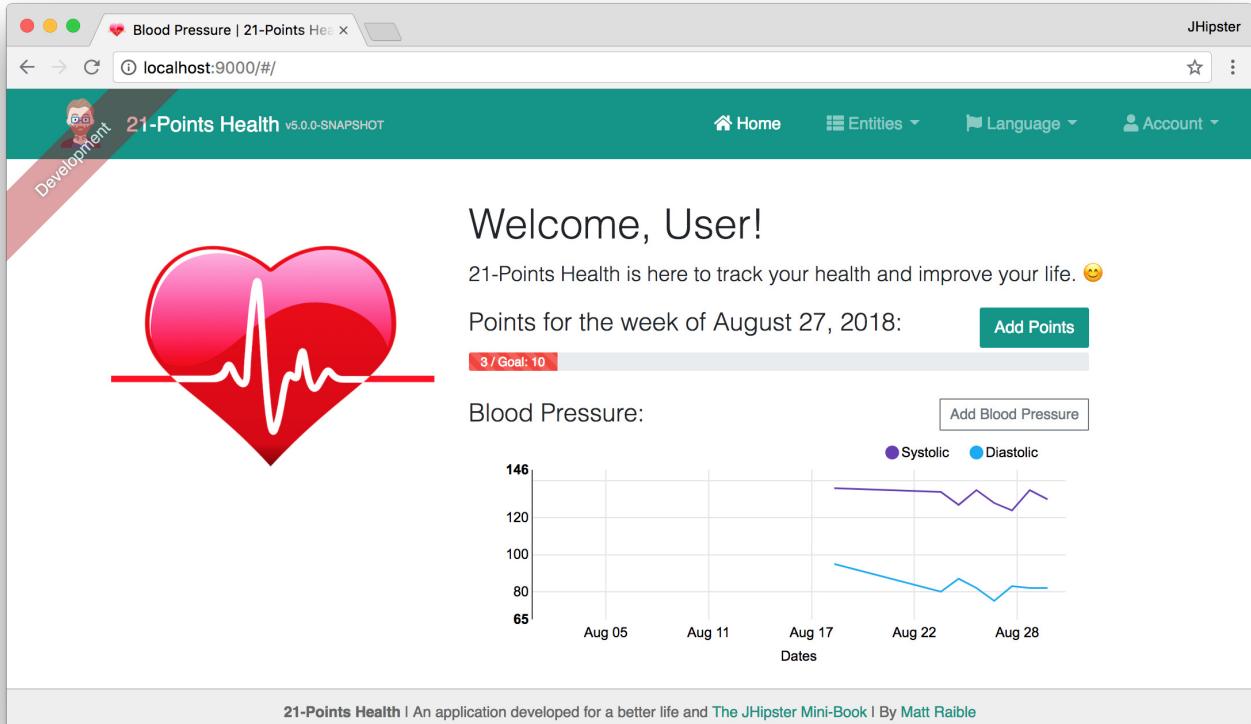


Figure 9. Chart of blood pressure during the last 30 days

I made similar changes to display weights for the last 30 days as a chart.

Lines of code

After finishing the MVP (minimum viable product) of 21-Points Health, I did some quick calculations to see how many lines of code JHipster produced. You can see from the graph below that I only had to write 2,080 lines of code. JHipster did the rest for me, generating 91.2% of the code in my project!

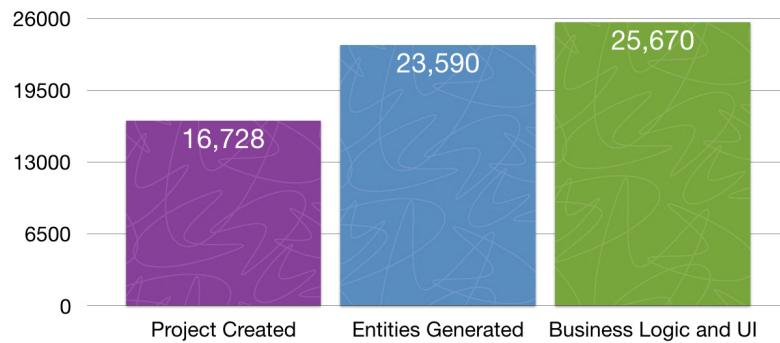


Figure 10. Project lines of code

To drill down further, I made a graph of the top three languages in the project: Java, TypeScript, and HTML.

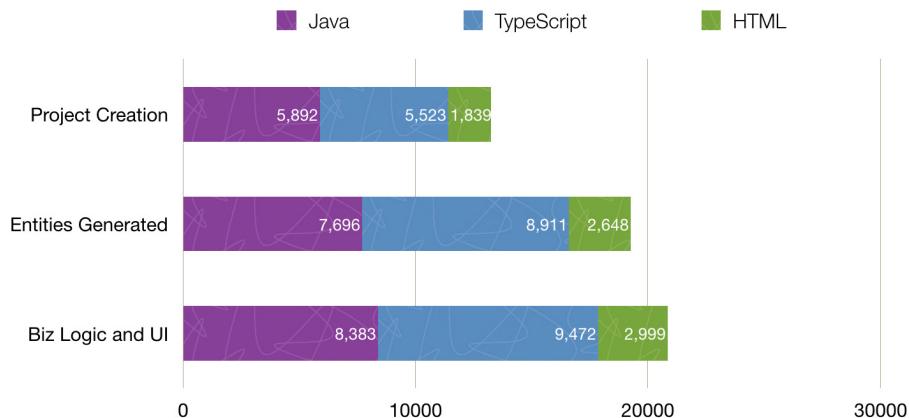


Figure 11. Project lines of code by language

The amount of code I had to write in each language was 561 lines of TypeScript, 687 lines of Java, and 351 lines of HTML. The other 481 lines were JSON (135), XML (187), Sass (80), YAML (54), CSS (12), Markdown (6), Groovy (4), and Bourne Shell (3).

Wahoo! Thanks, JHipster!

Testing

You probably noticed that a lot of the Java code I wrote was for the tests. I felt that these tests were essential to prove that the business logic I implemented was correct. It's never easy to work with dates but Java 8's Date-Time API greatly simplified it and Spring Data JPA made it easy to write "between date" queries.

I believe TDD (test-driven development) is a great way to write code. However, when developing UIs, I tend to make them work before writing tests. It's usually a very visual activity and, with the aid of Browsersync, there's rarely a delay before you see your changes. I like to write unit tests for my Angular components and directives using [Jasmine](#) and I like to write integration tests with [Protractor](#).

I did not show any UI tests in this section, but JHipster generated a bunch for me. Running `yarn test --coverage` shows 78.98% of lines are covered in the UI!

Deploying to Heroku

JHipster ships with support for deploying to Cloud Foundry, Heroku, Kubernetes, Microsoft Azure, OpenShift, Rancher, AWS, and Boxfuse. I used Heroku to deploy my application to the cloud because I'd worked with it before. When you prepare a JHipster application for production, it's recommended to use the pre-configured "prod" profile. With Gradle, you can package your application by specifying this profile when building.

```
./gradlew -Pprod bootWar
```

The command looks similar when using Maven.

```
./mvnw -Pprod package
```

The production profile is used to build an optimized JavaScript client. You can invoke this using webpack by running `yarn webpack:prod`. The production profile also configures gzip compression with a servlet filter, cache headers, and monitoring via [Metrics](#). If you have a [Graphite](#) server configured in your `application-prod.yml` file, your application will automatically send metrics data to it.

To deploy 21-Points Health, I logged in to my Heroku account. I already had the [Heroku CLI](#) installed.



I first deployed to Heroku after creating the application, meaning that I had a default JHipster application with no entities.

```
$ heroku login
Enter your Heroku credentials.
Email: matt@raibledesigns.com
Password (typing will be hidden):
Authentication successful.
```

I ran `jhipster heroku` as recommended in the [Deploying to Heroku](#) documentation. I tried using the name “21points” for my application when prompted.

```
$ jhipster heroku
Heroku configuration is starting
? Name to deploy as: 21points
? On which region do you want to deploy ? us
? Which type of deployment do you want ? Git (compile on Heroku)
```

Using existing Git repository

Heroku CLI deployment plugin already installed

```
Creating Heroku application and setting up node environment
{ Error: Command failed: heroku create 21-points
Creating 21-points... !
    Name must start with a letter and can only contain lowercase letters,
    numbers, and dashes.
```

You can see my first attempt failed for the same reason that creating the initial JHipster app failed: it didn’t like that the app name started with a number. I tried again with “health”, but that failed, too, since a Heroku app with this name already existed. Finally, I settled on “health-by-points” as the application name.

```
$ jhipster heroku
Using JHipster version installed locally in current project's node_modules
Executing jhipster:heroku
```

Options:

```
Heroku configuration is starting
? Name to deploy as: health-by-points
? On which region do you want to deploy ? us
```

Using existing Git repository

Heroku CLI deployment plugin already installed

Creating Heroku application and setting up node environment

```
https://health-by-points.herokuapp.com/ | https://git.heroku.com/health-by-points.git
```

Provisioning addons

Created Elasticsearch addon

Created Database addon

Creating Heroku deployment files

```
create src/main/resources/config/bootstrap-heroku.yml
create src/main/resources/config/application-heroku.yml
create Procfile
create gradle/heroku.gradle
```

conflict build.gradle

? Overwrite build.gradle? overwrite this and all others

force build.gradle

Skipping build

Updating Git repository

```
git add .
git commit -m "Deploy to Heroku" --allow-empty
```

Configuring Heroku

Deploying application

remote: Compressing source files... done.

remote: Building source:

...

remote: BUILD SUCCESSFUL in 4m 30s

remote: 8 actionable tasks: 7 executed, 1 up-to-date

remote: -----> Discovering process types

remote: Procfile declares types -> web

remote:

remote: -----> Compressing...

remote: Done: 141.3M

remote: -----> Launching...

remote: Released v6

remote: <https://health-by-points.herokuapp.com/> deployed to Heroku

remote:

remote: Verifying deploy... done.

To <https://git.heroku.com/health-by-points.git>

* [new branch] HEAD -> master

I was pumped to see that this process worked and that my application was available at <http://health-by-points.herokuapp.com>. I quickly changed the default passwords for **admin** and **user** to make things more secure.

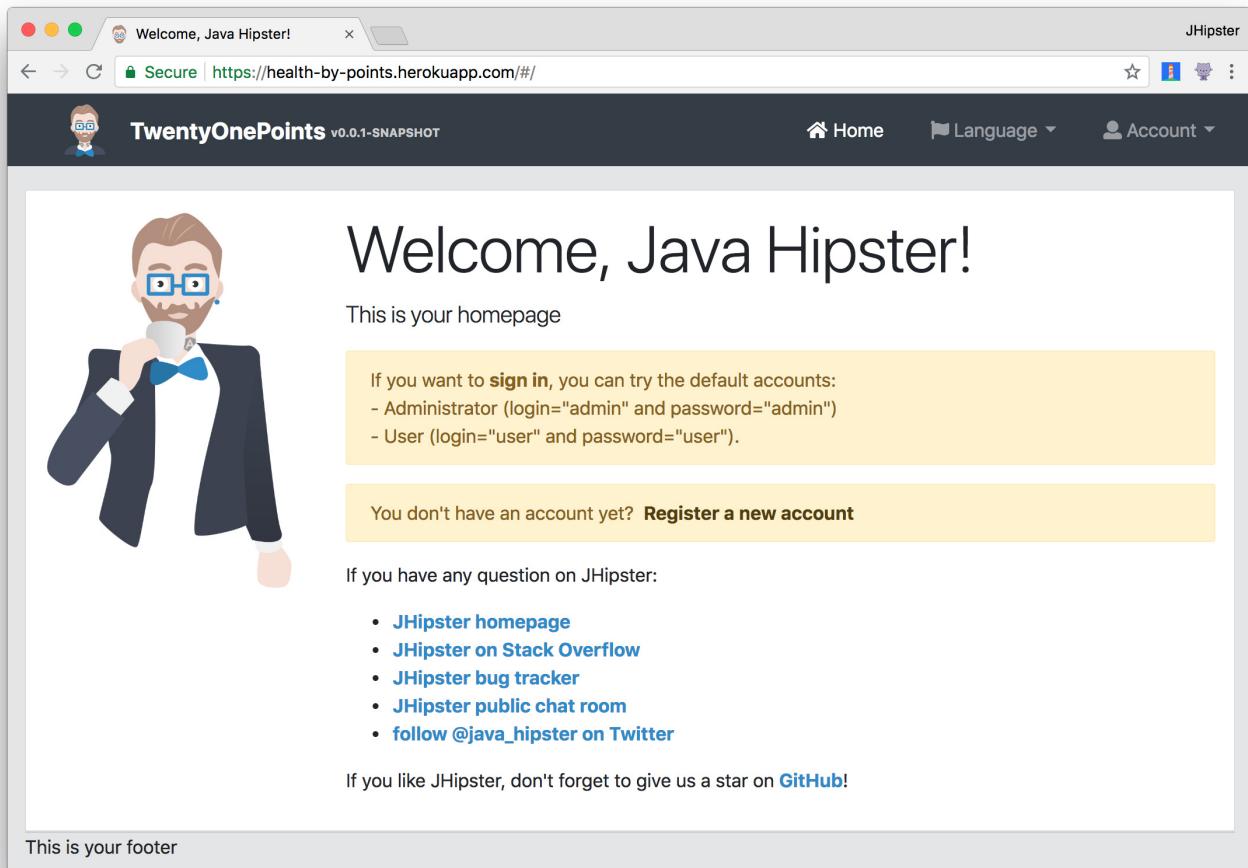


Figure 12. First deployment to Heroku

Next, I bought the 21-points.com domain from [Google Domains](#). To configure this domain for Heroku, I ran `heroku domains:add`.

```
$ heroku domains:add www.21-points.com
Adding www.21-points.com to health-by-points... done
!   Configure your app's DNS provider to point to the DNS Target www.21-points.com
!   For help, see https://devcenter.heroku.com/articles/custom-domains
```

I read the [documentation](#), then went to work configuring DNS settings on Google Domains. I configured a subdomain forward of:

```
21-points.com → http://www.21-points.com
```

I also configured a custom resource record with a CNAME to point to `health-by-points.herokuapp.com`.

Table 1. Custom resource record on Google Domains

Name	Type	TTL	Data
*	CNAME	1h	health-by-points.herokuapp.com

This was all I needed to get my JHipster application running on Heroku. For subsequent deployments, I ran `jhipster heroku` again, or used `git push heroku master`.

JAR Deployments to Heroku

If you use JAR deployments with Heroku, in addition to using `jhipster heroku` you can redeploy your application using `heroku-cli-deploy`. Use the following command to install this plugin.

```
heroku plugins:install heroku-cli-deploy
```

After that, you can package your JHipster project for production and deploy it. Using Gradle, it looks like this.

```
./gradlew -Pprod bootWar
heroku war:deploy build/libs/*war --app health-by-points
```

With Maven, the commands look slightly different:

```
./mvnw package -Pprod
heroku war:deploy target/*war --app health-by-points
```

Elasticsearch on Heroku

To prove everything was working on Heroku, I tried registering a new user. I received an error that appeared to come from Elasticsearch.

```
2018-08-27T21:15:27.588565+00:00 app[web.1]: 2018-08-27 21:15:27.587 ERROR 4 --- [ XNIO-2 task-15]
o.z.p.spring.web.advice.AdviceTrait      : Internal Server Error
2018-08-27T21:15:27.588578+00:00 app[web.1]:
2018-08-27T21:15:27.588581+00:00 app[web.1]: org.elasticsearch.client.transport.NoNodeAvailableException:
None of the configured nodes are available: [{#transport#-1}{7Du0TB4WQ_6crjWF01iJVQ}{localhost}{127.0.0.1:9300}]
2018-08-27T21:15:27.588583+00:00 app[web.1]: at org.elasticsearch.client.transport.TransportClientNodesService
.ensureNodesAreAvailable(TransportClientNodesService.java:347)
2018-08-27T21:15:27.588585+00:00 app[web.1]: at org.elasticsearch.client.transport.TransportClientNodesService.execute
(TransportClientNodesService.java:245)
```

I [created an issue](#) in the JHipster project saying that Elasticsearch doesn't work out of the box with Heroku. I worked with Joe Kutner of Heroku to come up with a solution and fix.

The fix for Elasticsearch on Heroku was released in JHipster 5.3.0, so I upgraded using the [upgrade sub-generator](#).

After the command finished, I committed changes to `yarn.lock`, ran `git push`, and deployed to Heroku again using `jhipster heroku`.

Mail on Heroku

After making this change, I repackaged and redeployed. This time, when I tried to register, I received an error when my `MailService` tried to send me an activation e-mail.

```
2018-08-27T21:26:12.193734+00:00 app[web.1]: 2017-08-14 21:26:12.193  WARN 4 --- [ints-Executor-2]
org.jhipster.health.service.MailService : Email could not be sent to user 'mraible@gmail.com':
Mail server connection failed; nested exception is com.sun.mail.util.MailConnectException:
Couldn't connect to host, port: localhost, 25; timeout -1;
2018-08-27T21:26:12.193748+00:00 app[web.1]: nested exception is:
2018-08-27T21:26:12.193751+00:00 app[web.1]: java.net.ConnectException: Connection refused
(Connection refused). Failed messages: com.sun.mail.util.MailConnectException: Couldn't connect
to host, port: localhost, 25; timeout -1;
```

I'd used Heroku's [SendGrid](#) for e-mail in the past, so I added it to my project.

```
$ heroku addons:create sendgrid
Creating giving-softly-5465... done, (free)
Adding giving-softly-5465 to health-by-points... done
Setting SENDGRID_PASSWORD, SENDGRID_USERNAME and restarting health-by-points... done, v17
Use 'heroku addons:docs sendgrid' to view documentation.
```

Then I updated `application-prod.yml` to use the configured `SENDGRID_PASSWORD` and `SENDGRID_USERNAME` environment variables for mail, as well as to turn on authentication.

src/main/resources/config/application-prod.yml

```
mail:
  host: smtp.sendgrid.net
  port: 587
  username: ${SENDGRID_USERNAME}
  password: ${SENDGRID_PASSWORD}
  protocol: smtp
  properties:
    tls: false
    auth: true
```

I also changed the `jhipster.mail.*` properties further down in this file.

```
mail:
  from: app@21-points.com
  base-url: http://www.21-points.com
```

After repackaging and redeploying, I used the built-in health-checks feature of my application to verify that everything was configured correctly.

Monitoring and analytics

JHipster generates the code necessary for Google Analytics in every application's `src/main/webapp/index.html` file. I chose not to enable this just yet, but I hope to eventually. I already have a Google Analytics account, so it's just a matter of creating a new account for `www.21-points.com`, copying the account number, and modifying the following section of `index.html`:

src/main/webapp/index.html

```
<!-- Google Analytics: uncomment and change UA-XXXXX-X to be your site's ID.
<script>
  (function(b,o,i,l,e,r){b.GoogleAnalyticsObject=l;b[l]|| (b[l]=
  function(){(b[l].q=b[l].q||[]).push(arguments)});b[l].l=+new Date;
  e=o.createElement(i);r=o.getElementsByTagName(i)[0];
  e.src='//www.google-analytics.com/analytics.js';
  r.parentNode.insertBefore(e,r)}(window,document,'script','ga'));
  ga('create','UA-XXXXX-X');ga('send','pageview');
</script>-->
```

I've used [New Relic](#) to monitor my production applications in the past. There is a free [New Relic add-on](#) for Heroku. Heroku's [New Relic APM](#) describes how to set things up if you're letting Heroku do the build for you (meaning, you deploy with `git push heroku master`). However, if you're using the heroku-

deploy plugin, it's a bit different.

For that, you'll first need to manually download the New Relic agent, as well as a `newrelic.yml` license file, and put them in the root directory of your project. Then you can run a command like:

```
heroku war:deploy build/libs/*war --includes newrelic.jar:newrelic.yml
```

That will include the JAR in the slug. Then you'll need to modify your Procfile to include the `javaagent` argument:

```
web: java -javaagent:newrelic.jar $JAVA_OPTS -Xmx256m -jar build/libs/*.war ...
```

Securing user data

After running the 5.0 version of 21-Points Health on Heroku for a couple weeks, someone reported [an issue with security on GitHub](#). They pointed out that if you searched, you could see another user's data. I also discovered you could edit data based on the URL too.

To fix this data leakage, I enhanced the Java code so it only allows users that own an entity to edit it. Here's some sudo-code to show the logic:

```
Optional<Points> points = pointsRepository.findById(id);
if (points.isPresent() && <user doesn't match current user>) {
    return new ResponseEntity<>("error.http.403", HttpStatus.FORBIDDEN);
}
return ResponseUtil.wrapOrNotFound(points);
```

See [pull request #52](#) for all the changes that I needed to make in resource classes and their tests.

Continuous integration and deployment

After generating entities for this project, I wanted to configure a continuous-integration (CI) server to build/test/deploy whenever I checked in changes to Git. I chose [Jenkins](#) for my CI server and used the simplest configuration possible: I downloaded `jenkins.war` to `/opt/tools/jenkins` on my MacBook Pro. I started it with the following command.

```
java -jar jenkins.war --httpPort=9000
```

JHipster has good documentation on [setting up CI on Jenkins 2](#) and [deploying to Heroku](#). It also has a handy sub-generator to generate the config files needed for Jenkins. I ran `jhipster ci-cd` and watched

the magic happen.

```
$ jhipster ci-cd
Using JHipster version installed locally in current project's node_modules
Executing jhipster:ci-cd
Options:
  ✈ Welcome to the JHipster CI/CD Sub-Generator ✈
? What CI/CD pipeline do you want to generate? Jenkins pipeline
? Would you like to perform the build in a Docker container ? No
? Would you like to send build status to GitLab ? No
? What tasks/integrations do you want to include ? Deploy to *Heroku* (requires HEROKU_API_KEY set on CI service)
? *Heroku*: name of your Heroku Application ? health-by-points
  create Jenkinsfile
  create src/main/docker/jenkins.yml
  create src/main/resources/idea.gds
```

After I generated these files, I checked them in and pushed to Bitbucket.

Jenkins Options

When choosing Jenkins, you can also select the following options for tasks/integrations:

- Deploy artifact to Artifactory.
- Analyze code with Sonar.
- Build and publish a Docker image.

To log in to Jenkins, I navigated to <http://localhost:9000>. I copied the password from the startup log file and pasted into the unlock Jenkins page.

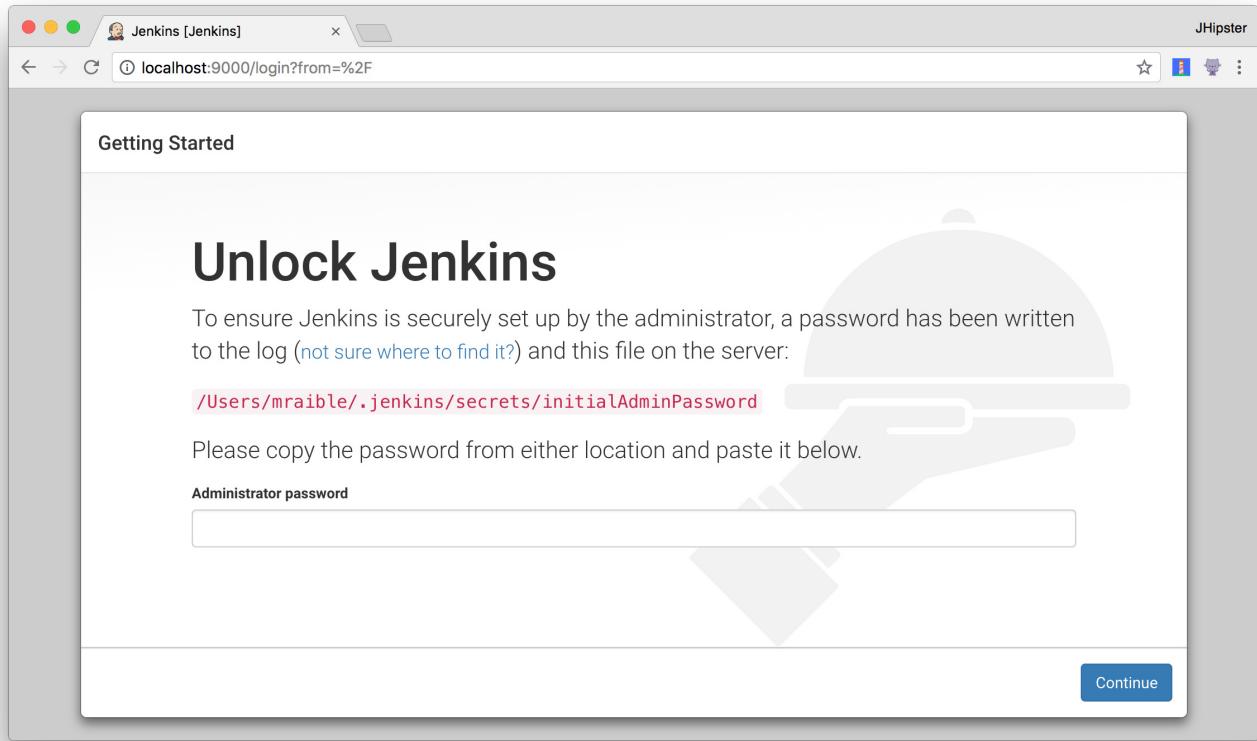


Figure 13. Unlock Jenkins

Next, I chose to install selected plugins and waited while everything installed.

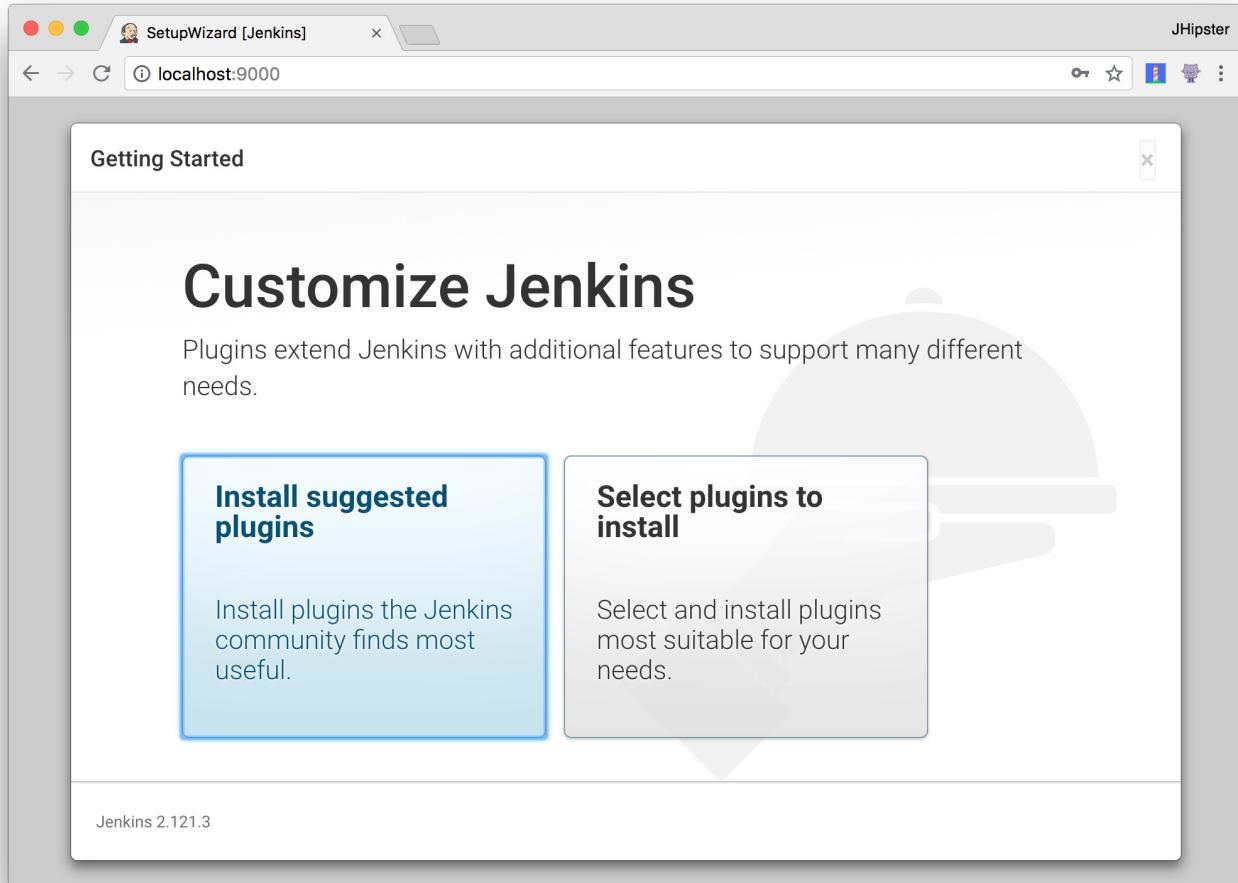


Figure 14. Customize Jenkins

I created a new job called "21-points" with a Pipeline script from SCM. I configured a "Poll SCM" build trigger with a schedule of `H/5 * * * *`. After saving the job, I confirmed it ran successfully.

The screenshot shows the Jenkins interface for a pipeline named 'Pipeline 21-points'. On the left, there's a sidebar with various Jenkins-related links like Back to Dashboard, Status, Changes, Build Now, Delete Pipeline, Configure, Full Stage View, Rename, Pipeline Syntax, and Git Polling Log. Below this is the 'Build History' section, which shows a single build (#1) from Aug 27, 2018, at 9:25 PM. The build status is 'No Changes'.

The main content area is titled 'Pipeline 21-points' and displays the 'Stage View'. It shows a timeline of stages: checkout, check java, clean, install tools, backend tests, frontend tests, packaging, and deployment. The 'checkout' stage took 1s, 'check java' took 309ms, 'clean' took 5s, 'install tools' took 9s, 'backend tests' took 1min 13s, 'frontend tests' took 49s, 'packaging' took 2min 21s, and 'deployment' took 3min 41s. The entire pipeline took approximately 26s.

Below the stage view, there's a 'Latest Test Result' section indicating 'no failures'. At the bottom, there's a 'Permalinks' section with a link to the last build.

Figure 15. Jenkins build #1



It's possible the `deployment` stage will fail for you the first time. If this happens, stop Jenkins, run `heroku login`, then restart Jenkins.

I modified this file to add a `protractor tests` stage to run all the Protractor tests. I checked in my changes to trigger another build.

Jenkinsfile

```
#!/usr/bin/env groovy

node {
    stage('checkout') {
        checkout scm
    }

    stage('check java') {
        sh "java -version"
    }

    stage('clean') {
        sh "chmod +x gradlew"
    }
}
```

```
sh "./gradlew clean --no-daemon"
}

stage('install tools') {
    sh "./gradlew yarn_install -PnodeInstall --no-daemon"
}

stage('backend tests') {
    try {
        sh "./gradlew test -PnodeInstall --no-daemon"
    } catch(err) {
        throw err
    } finally {
        junit '**/build/**/TEST-*.xml'
    }
}

stage('frontend tests') {
    try {
        sh "./gradlew yarn_test -PnodeInstall --no-daemon"
    } catch(err) {
        throw err
    } finally {
        junit '**/build/test-results/jest/TESTS-*.xml'
    }
}

stage('protractor tests') {
    sh '''./gradlew &
bootPid=$!
sleep 60s
yarn e2e
kill $bootPid
''''
}
}

stage('packaging') {
    sh "./gradlew bootWar -x test -Pprod -PnodeInstall --no-daemon"
    archiveArtifacts artifacts: '**/build/libs/*.war', fingerprint: true
}

stage('deployment') {
    sh "./gradlew deployHeroku --no-daemon"
}
}
```

I was pumped to see all the stages in my pipeline pass.

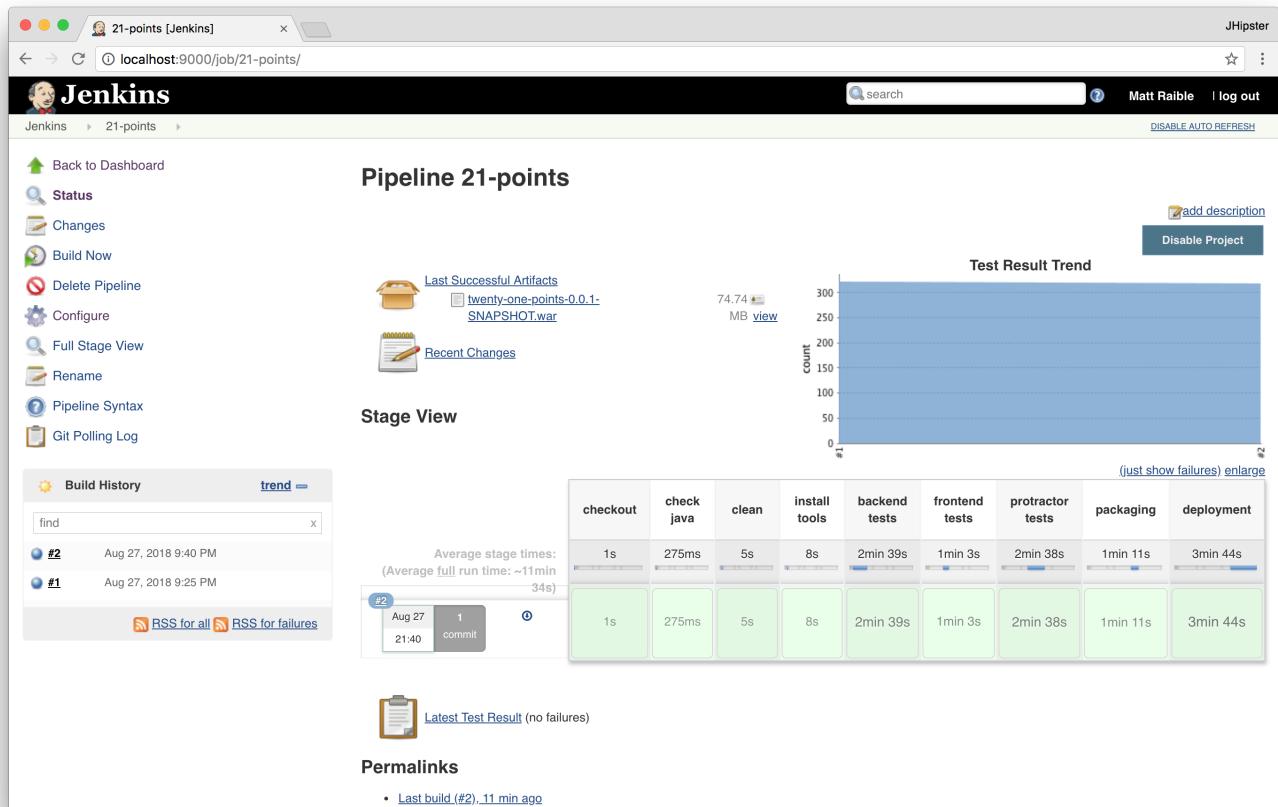


Figure 16. Jenkins success!

When working on this project, I'd start Jenkins and have it running while I checked in code. I did not install it on a server and leave it running continuously. My reason was simple: I was only coding in bursts and didn't need to waste computing cycles or want to pay for a cloud instance to run it.

Code quality

When I finished developing the app, I wanted to ensure that I had good code quality and that things were well tested. JHipster generates apps with high code quality by default. Code quality is analyzed using Sonar, which is automatically configured by JHipster. The "code quality" metric is determined by the percentage of code that is covered by tests. To see the code quality for my finished app, I started all the Docker containers that the prod profile needs, plus Sonar.

```
docker-compose -f src/main/docker/elasticsearch.yml up -d
docker-compose -f src/main/docker/postgresql.yml up -d
docker-compose -f src/main/docker/sonar.yml up -d
```

Then I ran all the tests and the `sonarqube` task.

```
/gradlew -Pprod clean test sonarqube
```

Once this process completed, an analysis of the project was available on the Sonar dashboard at <http://127.0.0.1:9001>. 21-Points Heath is a triple-A-rated app! Not bad, eh?

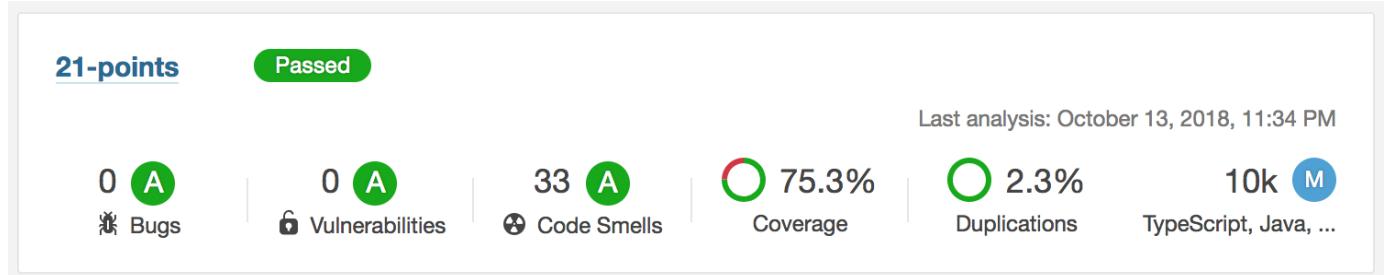


Figure 17. Sonar results

Progressive web apps

Progressive web apps, aka PWAs, are the best way for developers to make their web apps load faster and perform better. In a nutshell, PWAs are websites that use recent web standards to allow for installation on a user's computer or device, and deliver an app-like experience to those users.

To be a PWA requires three features:

1. The app must be served over HTTPS.
2. The app must register a service worker so it can cache requests and work offline.
3. The app must have a web-app manifest with installation information and icons.

For HTTPS, you can use JHipster's "tls" profile for localhost or (even better) deploy it to production!



To use HTTPS on localhost with Gradle, run `./gradlew -Ptls` for the back end and `yarn start-tls` for the front end.

Cloud providers like Heroku and Cloud Foundry will provide you with HTTPS out of the box, but they won't force it. To force HTTPS, I modified `src/main/java/org/jhipster/health/config/SecurityConfiguration.java` and added a rule to force a secure channel when an `X-Forwarded-Proto` header is sent.

```

@Override
public void configure(HttpSecurity http) throws Exception {
    http
        ...
        .and()
            .apply(securityConfigurerAdapter())
        .and()
            .requiresChannel()
            .requestMatchers(r -> r.getHeader("X-Forwarded-Proto") != null)
            .requiresSecure();
}

```

The [workbox-webpack-plugin](#) is configured already for generating a service worker, but it only works when running your app with a production profile. This is nice because it means your data isn't cached in the browser when you're developing.

To register a service worker, I modified [src/main/webapp/index.html](#) and uncommented the following block of code.

```

<script>
    if ('serviceWorker' in navigator) {
        window.addEventListener('load', function() {
            navigator.serviceWorker.register('/service-worker.js')
                .then(function () {
                    console.log('Service Worker Registered');
                });
        });
    }
</script>

```

The final feature — a web-app manifest — is already included at [src/main/webapp/manifest.webapp](#). It defines an app name, colors, and icons.

After making these changes, I redeployed 21-Points Health to production and used [Lighthouse](#) to perform an analysis. You can see the results in the screenshot below.

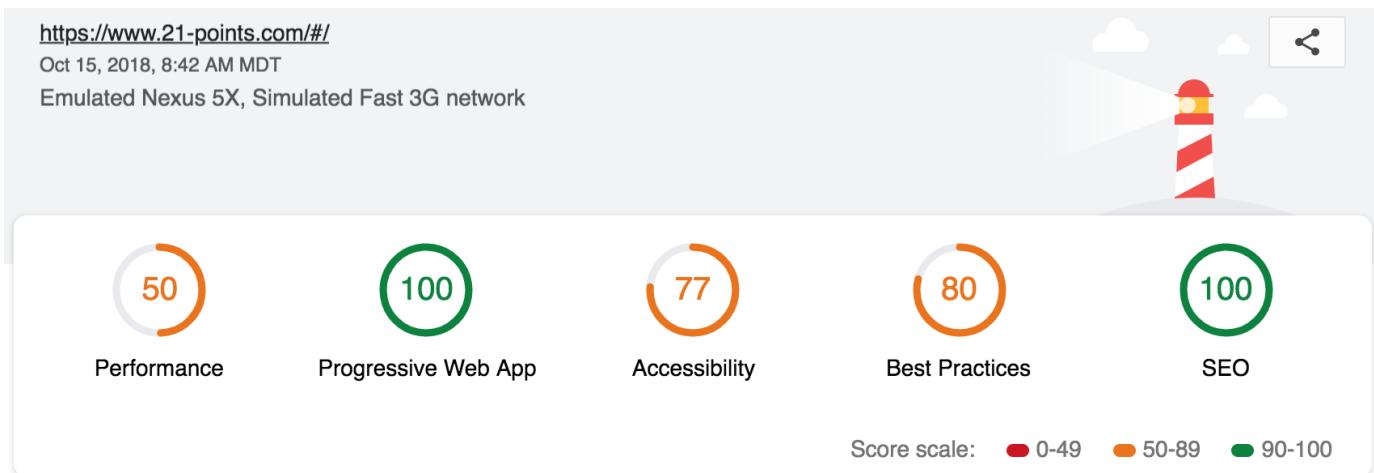


Figure 18. Lighthouse analysis

You can see there's still some performance improvements that need to be made. Rather than address those here, I've [created an issue](#) you can subscribe to.

Source code

After getting this application into a good enough state, I moved it from Bitbucket to GitHub and made it available as an open-source project. You can find the source code for 21-Points Health at <https://github.com/mraible/21-points>.

Upgrading 21-Points Health

After I finished developing the MVP of 21-Points Health with JHipster 5.3.0, I wanted to upgrade it to the latest release. I used the [upgrade sub-generator](#), and upgraded it to 5.3.4, then again to 5.4.2.



If the upgrade sub-generator doesn't work for you when upgrading JHipster, you can manually upgrade. You'll want to make sure your project is checked into source control first. Then, create a new branch and run `rm -rf *` in your project, followed by `jhipster --with-entities`. After this process completes, use your favorite tool (e.g., IntelliJ IDEA) to compare changed files and restore your custom code.

If you're reading this and notice that 21-Points Health is using a version newer than 5.4.2, it's likely because I upgraded again.

Summary

This section showed you how I created a health-tracking web application with JHipster. It walked you through upgrading to the latest release of JHipster and how to generate code with `jhipster entity`. You learned how to do test-first development when writing new APIs and how Spring Data JPA makes it easy to add custom queries. You also saw how to reuse existing components on different pages, how to

add methods to client services, and how to manipulate data to display pretty charts.

After modifying the application to look like my UI mockups, I showed you how to deploy to Heroku and some common issues I encountered along the way. Finally, you learned how to use Jenkins to build, test, and deploy a Gradle-based JHipster project. I highly recommend doing something similar shortly after you've created your project and verified that it passes all tests.

In the next chapter, I'll explain JHipster's UI components in more detail. Angular, Bootstrap, webpack, Sass, WebSockets, and Browsersync are all packed in a JHipster application, so it's useful to dive in and learn a bit more about these technologies.

PART TWO

JHipster's UI components

A modern web application has many UI components. It likely has some sort of model-view-controller (MVC) framework as well as a CSS framework, and tooling to simplify the use of these. With a web application, you can have users all over the globe, so translating your application into other languages might be important. If you're developing large amounts of CSS, you'll likely use a CSS pre-processor like Less or Sass. Then you'll need a build tool to refresh your browser, run your pre-processor, run your tests, minify your web assets, and prepare your application for production.

This section shows how JHipster includes all of these UI components for you and makes your developer experience a joyous one.

Angular

JHipster supports two UI frameworks: Angular and React. In the first two versions of this book, I showed you how to use AngularJS. In the last two versions (4.0 and 4.5), I showed you how to use Angular. Since this is a mini-book, I'm going to stick with showing Angular only. You can see from the following graphs that Angular and React are the most popular among JavaScript frameworks.

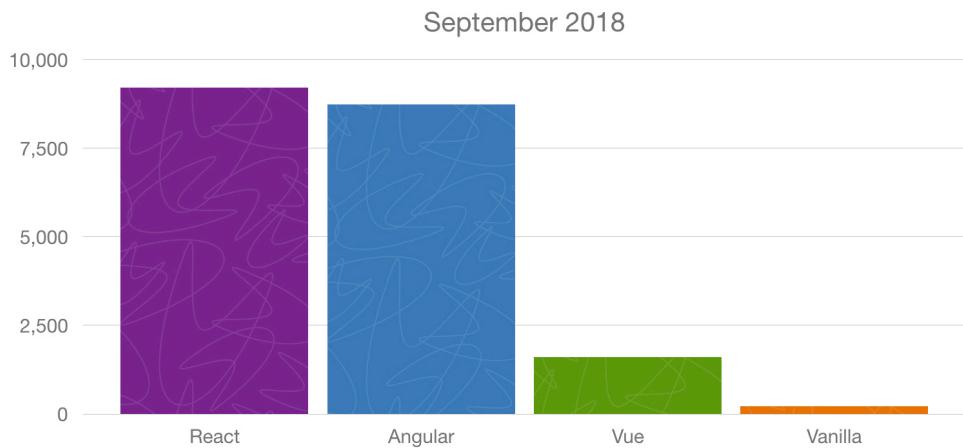


Figure 19. Jobs on Indeed, September 2018

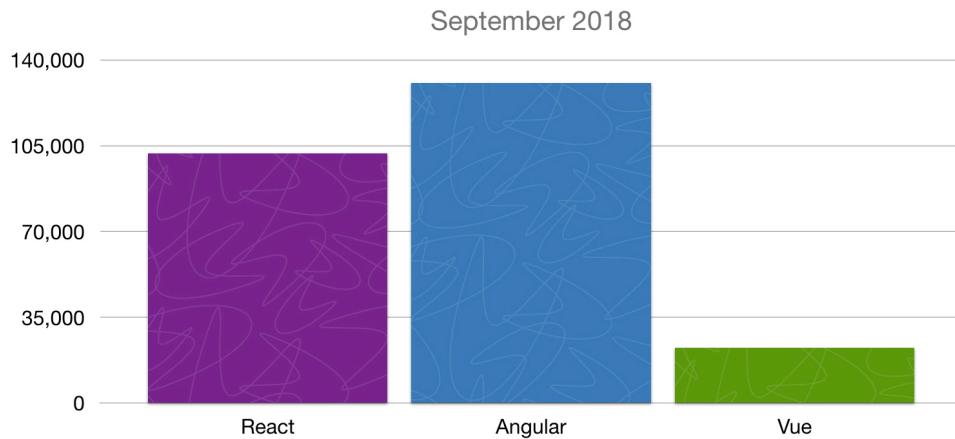


Figure 20. Stack Overflow Tags, September 2018

Angular is the default UI framework used by JHipster. It's written in TypeScript, compiles to JavaScript,

and just using it makes you a hipster! Like Struts in the early 2000s and Rails in the mid-2000s, Angular and other JavaScript frameworks have changed the way developers write web applications. Today, data is exposed via REST APIs and UIs are written in JavaScript (or TypeScript). As a Java developer, I was immediately attracted to Angular when I saw its separation of concerns. It recommended organizing your application into several different components:

- Components: Classes that retrieve data from services and expose it to templates.
- Services: Classes that make HTTP calls to a JSON API.
- Templates: HTML pages that display data. Use Angular directives to iterate over collections and show/hide elements.
- Pipes: Data-manipulation tools that can transform data (e.g. uppercase, lowercase, ordering, and searching).
- Directives: HTML processors that allow components to be written. Similar to JSP tags.

History

AngularJS was started by Miško Hevery in 2009. He was working on a project that was using GWT. Three developers had been developing the product for six months, and Miško rewrote the whole thing in AngularJS in three weeks. At that time, AngularJS was a side project he'd created. It didn't require you to write much in JavaScript as you could program most of the logic in HTML. The GWT version of the product contained 17,000 lines of code. The AngularJS version was only 1,000 lines of code!

In October 2014, the AngularJS team announced they were building [Angular 2.0](#). The announcement led to a bit of upheaval in the Angular developer community. The API for writing Angular applications was going to change and it was to be based on a new language, AtScript. There would be no migration path and users would have to continue using 1.x or rewrite their applications for 2.x.

A new syntax was introduced that binds data to element properties, not attributes. The advantage of this syntax is it allows you to use any web component in an Angular app, not just those retrofitted to work with Angular.

```
<input type="text" [value]="firstName">
<button (click)="addPerson()">Add</button>
<input type="checkbox" [checked]="someProperty">
```

In March 2015, the Angular team [addressed community concerns](#), announcing that they would be using TypeScript over AtScript and that they would provide a migration path for Angular 1.x users. They also adopted semantic versioning and [recommended people call it "Angular" instead of Angular 2.0](#).

Angular 2.0 was released September 2016. Angular 4.0 was released March 2017. JHipster 4.6.0 was released July 6, 2017 and was the first release to contain production-ready Angular support. JHipster 5 uses Angular 6. Angular released version 7 during the production of this book.

You can find the Angular project at <https://angular.io>.

Basics

Creating a component that says "Hello World" with Angular is pretty simple.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`
})
export class AppComponent {
  name = 'World';
}
```

In this example, the `name` variable in the component maps to the value displayed in `{{name}}`. To make this component render on a page, you'll need a few more files: a module definition, a bootstrapping class, and an HTML file. A basic module definition contains component declarations, imports, providers, and a class to bootstrap.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Then you'll need a bootstrap file, typically named `main.ts`. This file bootstraps the module.

```

import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));

```

Finally, you'll need a basic HTML file that renders the component.

```

<html>
  <head>
    <title>Howdy</title>
  </head>
  <body>
    <my-app></my-app>
    <script src="path/to/compiled/javascript.js"></script>
  </body>
</html>

```

The MVC pattern is a common one for web frameworks to implement. With Angular, the model is represented by an object that you create or retrieve from a service. The view is a HTML template and the component is a class that sets variables to be read by the template.

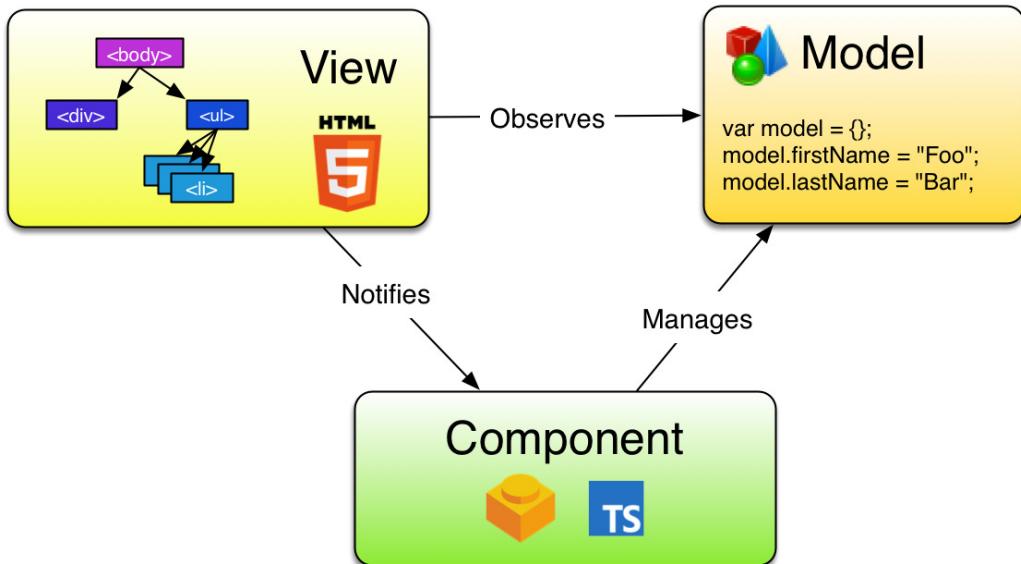


Figure 21. MVC in Angular

Below is a [SearchService](#) to fetch search results. It's expected that a JSON endpoint exists at `/api/search` on the same server.

SearchService

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class SearchService {

  constructor(private http: HttpClient) {}

  search(term): Observable<any> {
    return this.http.get('/api/search/${term}');
  }
}
```

An associated [SearchComponent](#) can be used to display the results from this service. Notice how you can use constructor injection to get a reference to the service.

SearchComponent

```
import { Component } from '@angular/core';
import { SearchService } from '../search.service';

@Component({
  selector: 'app-search',
  templateUrl: './search.component.html',
  styleUrls: ['./search.component.css']
})
export class SearchComponent {
  query: string;
  searchResults: Array<any>;

  constructor(private searchService: SearchService) {
    console.log('In Search Component...');
  }

  search(): void {
    this.searchService.search(this.query).subscribe(
      data => {
        this.searchResults = data;
      },
      error => console.log(error)
    );
  }
}
```



To see the JavaScript console in Chrome, use **Command+Option+J** in Mac OS X/macOS or **Control+Shift+J** in Windows or Linux.

To make this component available at a URL, you can use Angular's **Router** and specify the path in the module that includes the component.

AppModule

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';

import { AppComponent } from './app.component';
import { SearchComponent } from './search/search.component';
import { Routes, RouterModule } from '@angular/router';

const appRoutes: Routes = [
  { path: 'search', component: SearchComponent },
  { path: '', redirectTo: '/search', pathMatch: 'full' }
];

@NgModule({
  declarations: [
    AppComponent,
    SearchComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    RouterModule.forRoot(appRoutes)
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

In the main entry point of the app, `AppComponent` in this case, you'll need to specify the router outlet in a template.

app.component.html

```
<router-outlet></router-outlet>
```

The template for the `SearchComponent` can be as simple as a form with a button.

```
<h2>Search</h2>
<form>
  <input type="search" name="query" [(ngModel)]="query" (keyup.enter)="search()">
  <button type="button" (click)="search()">Search</button>
</form>
```

Now that you've seen the code, let's look at how everything works in the `SearchComponent`.

```
@Component({
  selector: 'app-search',
  templateUrl: './search.component.html',
  styleUrls: ['./search.component.css']
})
export class SearchComponent {
  query: string;
  searchResults: Array<any>;

  constructor(private searchService: SearchService) { ①
    console.log('In Search Component...');

  }

  search(): void { ②
    this.searchService.search(this.query).subscribe( ③
      data => {
        this.searchResults = data;
      },
      error => console.log(error)
    );
  }
}
```

① To inject `SearchService` into `SearchComponent`, simply add it as a parameter to the component's constructor. TypeScript automatically makes constructor dependencies available as class variables.

② `search()` is a method that's called from the HTML's `<input>` and `<button>`, wired up using the `(keyup.enter)` and `(click)` event handlers.

③ `this.query` is a variable that's wired to `<input>` using the `[(ngModel)]` directive. This syntax provides two-way binding so if you change the value in the component, it changes it in the rendered HTML. You can think of it this way: `[] ⇒ component to template` and `() ⇒ template to component`.

To make the aforementioned code work, you can generate a new Angular project using [Angular CLI](#). To install Angular CLI, you can use npm.

```
npm i -g @angular/cli
```

Then generate a new application using `ng new`. When prompted to install Angular routing, type “Y”. For the stylesheet format, choose “CSS” (the default).

```
ng new ng-demo
```

This creates all the files you need for a basic app, installs dependencies, and sets up a build system for compiling your TypeScript code to JavaScript.

You can generate the `SearchService` using `ng generate service search` (or `ng g s search`).

```
$ ng g s search
CREATE src/app/search.service.spec.ts (333 bytes)
CREATE src/app/search.service.ts (135 bytes)
```

And you can generate the `SearchComponent` using `ng generate component search` (or `ng g c search`).

```
$ ng g c search
CREATE src/app/search/search.component.css (0 bytes)
CREATE src/app/search/search.component.html (25 bytes)
CREATE src/app/search/search.component.spec.ts (628 bytes)
CREATE src/app/search/search.component.ts (269 bytes)
UPDATE src/app/app.module.ts (396 bytes)
```

Does your API return data like the following?

```
[  
  {  
    "id": 1,  
    "name": "Peyton Manning",  
    "phone": "(303) 567-8910",  
    "address": {  
      "street": "1234 Main Street",  
      "city": "Greenwood Village",  
      "state": "CO",  
      "zip": "80111"  
    }  
  },  
  {  
    "id": 2,  
    "name": "Demaryius Thomas",  
    "phone": "(720) 213-9876",  
    "address": {  
      "street": "5555 Marion Street",  
      "city": "Denver",  
      "state": "CO",  
      "zip": "80202"  
    }  
  },  
  {  
    "id": 3,  
    "name": "Von Miller",  
    "phone": "(917) 323-2333",  
    "address": {  
      "street": "14 Mountain Way",  
      "city": "Vail",  
      "state": "CO",  
      "zip": "81657"  
    }  
  }  
]
```

If so, you could display it in the `search.component.html` template with Angular's `*ngFor` directive.

```

<table *ngIf="searchResults">
  <thead>
    <tr>
      <th>Name</th>
      <th>Phone</th>
      <th>Address</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let person of searchResults; let i=index">
      <td><a [routerLink]=["/edit", person.id]>{{person.name}}</a></td>
      <td>{{person.phone}}</td>
      <td>{{person.address.street}}<br/>
        {{person.address.city}}, {{person.address.state}} {{person.address.zip}}
      </td>
    </tr>
  </tbody>
</table>

```

To read a more in-depth example (including source code and tests) of building a search/edit application with Angular, see my [Angular and Angular CLI Tutorial](#).

Now that you've learned a bit about one of the hottest web frameworks on the planet, let's take a look at the most popular CSS framework: Bootstrap.

Bootstrap

[Bootstrap](#) is a CSS framework that simplifies the development of web applications. It provides a number of CSS classes and HTML structures that allow you to develop HTML user interfaces that look good by default. Not only that, but it's responsive by default, which means it works better (or even best) on a mobile device.

Bootstrap's grid system

Most CSS frameworks provide a grid system that allows you to position columns and cells in a respectable way. Bootstrap's powerful grid is built with containers, rows, and columns. It's based on the CSS3 flexible box, or flexbox. Flexbox is a layout mode intended to accommodate different screen sizes and different display devices. It's easier than using blocks to do layouts because it doesn't use floats, nor do the flex container's margins collapse with the margins of its content. CSS-Tricks has [A Complete Guide to Flexbox](#) that explains its concepts well.

The main idea behind the flex layout is to give the container the ability to alter its items' width/height (and order) to best fill the available space, mostly to accommodate all kinds of display devices and screen sizes. A flex container expands items to fill available free space or shrinks them to prevent overflow.

Bootstrap's grid width varies based on viewport width. The table below shows how aspects of the grid system work across different devices.

	Extra small	Small	Medium	Large	Extra large
Max container width	None (auto)	540px	720px	960px	1140px
Class prefix	.col-	.col-sm-	.col-md-	.col-lg-	.col-xl-
# of columns	12				
Gutter width	30px (15px on each side)				
Nestable	Yes				
Column ordering	Yes				

A basic example of the grid is shown below.

```
<div class="row">
  <div class="col-md-3">.col-md-3 <!-- 3 columns on the left --></div>
  <div class="col-md-9">.col-md-9 <!-- 9 columns on the right --></div>
</div>
```

When rendered with Bootstrap's CSS, the above HTML looks as follows on a desktop. The minimum width of the container element on the desktop is set to 1200 px.



Figure 22. Basic grid on desktop

If you squish your browser to less than 1200 px wide or render this same document on a smaller screen, the columns will stack.



Figure 23. Basic grid on a mobile device

Bootstrap's grid can be used to align and position your application's elements, widgets, and features. It's helpful to understand a few basics if want to use it effectively.

- It's based on 12 columns.
- Just use "md" class and fix as needed.
- It can be used to size input fields.

Bootstrap's grid system has five tiers of classes: xs (portrait phones), sm (landscape phones), md (tablets), lg (desktops), and xl (large desktops). You can use nearly any combination of these classes to create more dynamic and flexible layouts. Below is an example of a grid that's a little more advanced.

Each tier of classes scales up, meaning that if you plan to set the same widths for xs and sm, you only need to specify xs.

```
<div class="row">
    <div class="col-xs-12 col-md-8">.col-xs-12 .col-md-8</div>
    <div class="col-xs-6 col-md-4">.col-xs-6 .col-md-4</div>
</div>
<div class="row">
    <div class="col-xs-6 col-md-4">.col-xs-6 .col-md-4</div>
    <div class="col-xs-6 col-md-4">.col-xs-6 .col-md-4</div>
    <div class="col-xs-6 col-md-4">.col-xs-6 .col-md-4</div>
</div>
<div class="row">
    <div class="col-xs-6">.col-xs-6</div>
    <div class="col-xs-6">.col-xs-6</div>
</div>
```

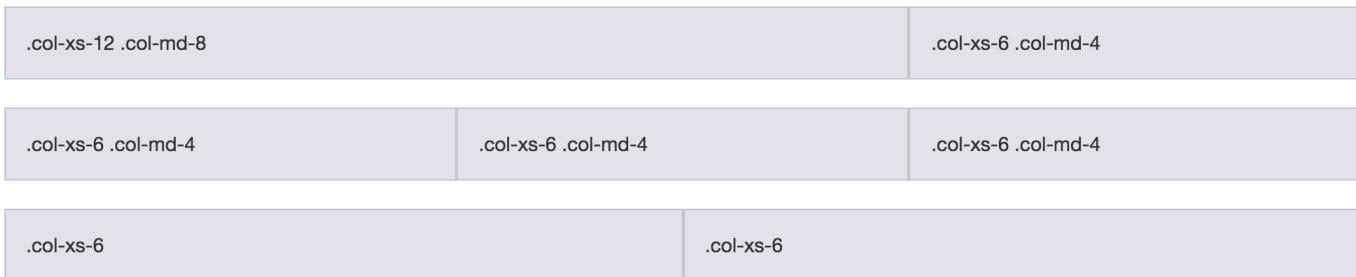


Figure 24. Advanced grid

You can use size indicators to specify breakpoints in the columns. Breakpoints indicate where a column wraps onto the next row. In the HTML above, “xs” and “md” are the size indicators (of course, “sm”, “lg”, and “xl” are the other options). Bootstrap uses the following media query ranges.

```
// Extra small devices (portrait phones, less than 576 px)
// No media query since this is the default in Bootstrap

// Small devices (landscape phones, 576 px and up)
@media (min-width: 576px) { ... }

// Medium devices (tablets, 768 px and up)
@media (min-width: 768px) { ... }

// Large devices (desktops, 992 px and up)
@media (min-width: 992px) { ... }

// Extra large devices (large desktops, 1200 px and up)
@media (min-width: 1200px) { ... }
```

If you’re using Sass, all Bootstrap’s media queries are available via Sass mixins:

```
// No media query necessary for xs breakpoint as it's effectively '@media (min-width: 0)
{ ... }'
@include media-breakpoint-up(sm) { ... }
@include media-breakpoint-up(md) { ... }
@include media-breakpoint-up(lg) { ... }
@include media-breakpoint-up(xl) { ... }

// Example: Hide starting at 'min-width: 0', and then show at the 'sm' breakpoint
.custom-class {
  display: none;
}
@include media-breakpoint-up(sm) {
  .custom-class {
    display: block;
  }
}
```

Responsive utility classes

Bootstrap also includes a number of utility classes that can be used to show and hide elements based on browser size, like `.d-[xs|sm|md|lg]-block` and `.d-[xs|sm|md|lg]-none`. There are no explicit "show" responsive utility classes; you make an element visible by simply not hiding it at that breakpoint size.

Bootstrap's classes for setting the display are names using the following format:

- `.d-{value}` for `xs`
- `.d-{breakpoint}-{value}` for `sm`, `md`, `lg`, and

The media queries effect screen widths with the given breakpoint or *larger*. For example, `.d-lg-none` sets `display: none` on both `lg` and `xl` screens.

The following example from 21-Points Health shows how to display a shorter heading on mobile and a larger one on bigger screens.

```
<div class="col-6 text-nowrap">
  <h4 class="mt-1 d-none d-md-inline"
      jhiTranslate="home.bloodPressure.title">Blood Pressure:</h4>
  <h4 class="mt-1 d-sm-none"
      jhiTranslate="home.bloodPressure.titleMobile">BP:</h4>
</div>
```

Forms

When you add Bootstrap's CSS to your web application, chances are it'll quickly start to look better. Typography, margins, and padding will look better by default. However, your forms might look funny, because Bootstrap requires a few classes on your form elements to make them look good. Below is an example of a form element.

```
<div class="form-group">
    <label for="description">Description</label>
    <textarea class="form-control" rows="4" name="description" id="description"></textarea>
</div>
```

Description

Figure 25. Basic form element

If you'd like to indicate that this form element is not valid, you'll need to modify the above HTML to display validation warnings.

```
<div class="form-group">
    <label for="description" class="control-label">Description</label>
    <textarea class="form-control is-invalid" rows="4" id="description" required></textarea>
    <span class="invalid-feedback">Description is a required field.</span>
</div>
```

Description

Description is a required field.

Figure 26. Form element with validation

CSS

When you add Bootstrap's CSS to a HTML page, the default settings immediately improve the typography. Your `<h1>` and `<h2>` headings become semi-bold and are sized accordingly. Your paragraph

margins, body text, and block quotes will look better. If you want to align text in your pages, `text-[left|center|right]` are useful classes. For tables, a `table` class gives them a better look and feel by default.

To make your buttons look better, Bootstrap provides `btn` and a number of `btn-*` classes.

```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-dark">Dark</button>
<button type="button" class="btn btn-link">Link</button>
```



Figure 27. Buttons

Components

Bootstrap ships with a number of components included. Some require JavaScript; some only require HTML5 markup and CSS classes to work. Its rich set of components have helped make it one of the most popular projects on GitHub. Web developers have always liked components in their frameworks. A framework that offers easy-to-use components often allows developers to write less code. Less code to write means there's less code to maintain!

Popular Bootstrap components include: dropdowns, button groups, button dropdowns, navbar, breadcrumbs, pagination, alerts, progress bars, and panels. Below is an example of a navbar with a dropdown.

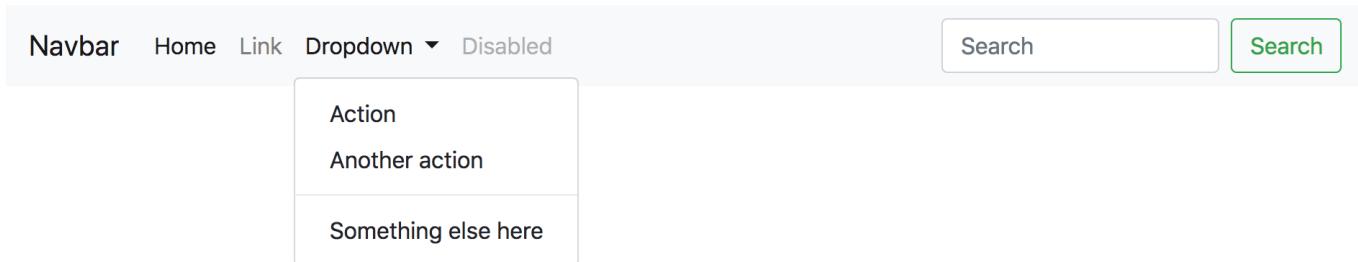


Figure 28. Navbar with dropdown

When rendered on a mobile device, everything collapses into a hamburger menu that can expand downward.

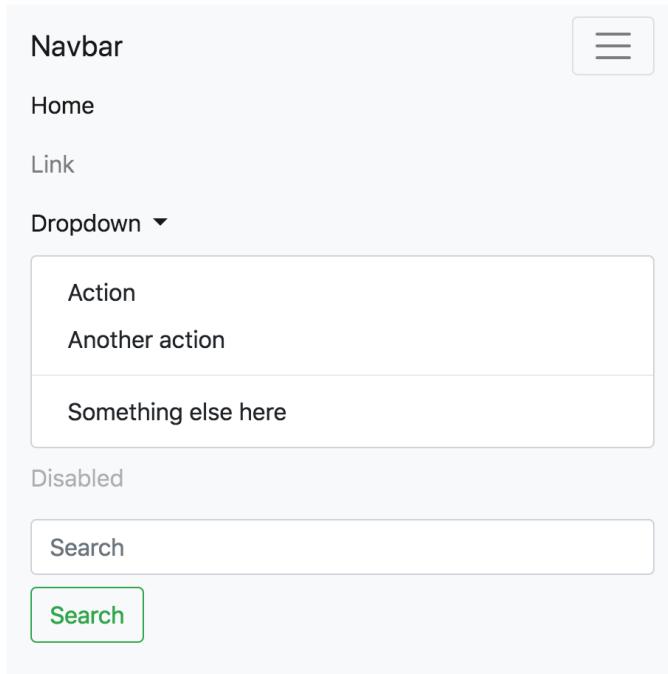


Figure 29. Navbar on mobile

This navbar requires quite a bit of HTML markup, not shown here for the sake of brevity. You can view this source online in [Bootstrap's documentation](#). A simpler example below shows the basic structure.

```

<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="#">Navbar</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse"
        data-target="#navbarSupportedContent">
        <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarSupportedContent">
        <ul class="navbar-nav mr-auto">
            <li class="nav-item active">
                <a class="nav-link" href="#">Home</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#">Link</a>
            </li>
            <li class="nav-item dropdown">
                <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown"
                    role="button" data-toggle="dropdown">
                    Dropdown
                </a>
                <div class="dropdown-menu" aria-labelledby="navbarDropdown">
                    <a class="dropdown-item" href="#">Action</a>
                    <a class="dropdown-item" href="#">Another action</a>
                    <div class="dropdown-divider"></div>
                    <a class="dropdown-item" href="#">Something else here</a>
                </div>
            </li>
            <li class="nav-item">
                <a class="nav-link disabled" href="#">Disabled</a>
            </li>
        </ul>
        <form class="form-inline">
            <input class="form-control" type="search" placeholder="Search">
            <button class="btn btn-outline-success" type="submit">Search</button>
        </form>
    </div>
</nav>

```

Alerts are useful for displaying feedback to the user. You can invoke differently colored alerts with different classes. You'll need to add an `alert` class, plus `alert-[success|info|warning|danger]` to indicate the colors.

```
<div class="alert alert-primary" role="alert">  
    This is a primary alert&mdash;check it out!  
</div>  
<div class="alert alert-secondary" role="alert">  
    This is a secondary alert&mdash;check it out!  
</div>  
<div class="alert alert-success" role="alert">  
    This is a success alert&mdash;check it out!  
</div>  
<div class="alert alert-danger" role="alert">  
    This is a danger alert&mdash;check it out!  
</div>  
<div class="alert alert-warning" role="alert">  
    This is a warning alert&mdash;check it out!  
</div>  
<div class="alert alert-info" role="alert">  
    This is a info alert&mdash;check it out!  
</div>  
<div class="alert alert-light" role="alert">  
    This is a light alert&mdash;check it out!  
</div>  
<div class="alert alert-dark" role="alert">  
    This is a dark alert&mdash;check it out!  
</div>
```

This renders alerts like the following.

A simple primary alert—check it out!

A simple secondary alert—check it out!

A simple success alert—check it out!

A simple danger alert—check it out!

A simple warning alert—check it out!

A simple info alert—check it out!

A simple light alert—check it out!

A simple dark alert—check it out!

Figure 30. Alerts

To make an alert closeable, you need to add an `.alert-dismissible` class and a close button.

```
<div class="alert alert-warning alert-dismissible fade show" role="alert">
  <strong>Warning!</strong> Better check yourself, you're not looking too good.
  <button type="button" class="close" data-dismiss="alert" aria-label="Close">
    <span aria-hidden="true">&times;</span>
  </button>
</div>
```

Warning! Better check yourself, you're not looking too good.



Figure 31. Closeable alert



To make the links in your alerts match the colors of the alerts, use `.alert-link`.

Icons

Icons have always been a big part of web applications. Showing the user a small image is often sexier

and hipper than plain text. Humans are visual beings and icons are a great way to spice things up. In the last several years, font icons have become popular for web development. Font icons are just fonts, but they contain symbols and glyphs instead of text. You can style, scale, and load them quickly because of their small size.

Bootstrap 3.x included the [Glyphicons Halflings](#) set of font icons. It's not normally free, but the font creator made it free for Bootstrap users. One of the [changes](#) for Bootstrap 4 is that it dropped the Glyphicons icon font. You can use [Font Awesome](#) and [GitHub Octicons](#) as a free alternatives to Glyphicons.

Font Awesome has 1,341 icons and is included by default in JHipster with [angular-fontawesome](#). It is often used to display eye candy on buttons.

```
<button class="btn btn-info"><fa-icon [icon]="'plus'"></fa-icon> Add</button>
<button class="btn btn-danger"><fa-icon [icon]="'times'"></fa-icon> Delete</button>
<button class="btn btn-success"><fa-icon [icon]="'pencil-alt'"></fa-icon> Edit</button>
```

You can see how the icons change color based on the font color defined for the element that contains them.



Figure 32. Buttons with icons

Customizing CSS

If you'd like to override Bootstrap classes in your project, the easiest thing to do is to put the override rule in a CSS file that comes after Bootstrap's CSS. Or you can modify `src/main/webapp/content/css/global.css` directly. If you're using Sass to compile `*.scss` files, you can modify `src/main/webapp/content/scss/global.scss` instead. Using Sass results in a much more concise authoring environment. Below is the default `vendor.scss` file that JHipster generates. You can see that it imports Bootstrap and I've added an import for Angular Calendar. Default Bootstrap rules are overridden in `src/main/webapp/content/scss/global.scss`.

`src/main/webapp/scss/vendor.scss`

```
/* after changing this file run 'yarn run webpack:build' */

/*****
put Sass variables here:
eg $input-color: red;
*****/

// Calendar styles
@import '~angular-calendar/scss/angular-calendar';
// Override Bootstrap variables
@import 'bootstrap-variables';
// Import Bootstrap source files from node_modules
@import '~bootstrap/scss/bootstrap';

/* jhipster-needle-scss-add-vendor JHipster will add new css style */
```

There's also a `src/main/webapp/content/scss/_bootstrap-variable.scss` file. You can modify this file to change the default Bootstrap settings like colors, border radius, etc.

Angular and Bootstrap

JHipster includes `ng-bootstrap` by default. This library provides Bootstrap 4 components that are powered by Angular instead of jQuery.

`Toastr` is a popular non-blocking notification library powered by jQuery. JHipster has an `alert.service.ts` that allows you to use Toastr-like alerts.

To use Toastr-like alerts in your project, modify `shared-libs.module.ts` to change `alertAsToast` to `true`.

`src/main/webapp/app/shared/shared-libs.module.ts`

```
@NgModule({
  imports: [
    NgbModule.forRoot(),
    NgJhipsterModule.forRoot({
      // set below to true to make alerts look like toast
      alertAsToast: true,
      i18nEnabled: true,
      defaultI18nLang: 'en'
    }),
    InfiniteScrollModule,
    CookieModule.forRoot(),
    FontAwesomeModule
  ],
  exports: [ ... ]
})
```

Popular alternatives to Bootstrap include [Angular Material](#) and [Ionic Framework](#). There is no support for these frameworks at this time. To integrate them would require that all templates to rewritten to include their classes instead of Bootstrap's. While possible, it'd be a lot of work to create and maintain.

Internationalization (i18n)

Internationalization (also called i18n because the word has 18 letters between “i” and “n”) is a first-class citizen in JHipster. Translating an application to another language is easiest if you put the i18n system in place at the beginning of a project. `ngx-translate` provides directives that make it easy to translate your application into multiple languages. You won't see this dependency in your JHipster project's `package.json` because it's included in the `ng-jhipster` project and wrapped in a `jhiTranslate` directive.

To use i18n in a JHipster project, you simply add a “jhiTranslate” attribute with a key.

```
<label for="username" jhiTranslate="global.form.username">Login</label>
```

The key references a JSON document, which will return the translated string. Angular will then replace the “First Name” string with the translated version.

JHipster allows you to choose a default language and translations when you first create a project. It stores the JSON documents for these languages in `src/main/webapp/i18n`. You can install additional languages using `yo jhipster:languages`. As of October 2018, JHipster supports 42 languages. You can also add a new language. To set the default language, modify `src/main/webapp/app/shared/shared-libs.module.ts` and its `defaultI18nLang` setting.

src/main/webapp/app/shared/shared-libs.module.ts

```
imports: [
  ...
  NgJhipsterModule.forRoot({
    // set below to true to make alerts look like toast
    alertAsToast: false,
    i18nEnabled: true,
    defaultI18nLang: 'en'
  }),
  ...
],
```

Sass

Sass stands for “syntactically awesome style sheets”. It’s a language for writing CSS with the goodies you’re used to using in modern programming languages, such as variables, nesting, mixins, and inheritance. Sass uses the `$` symbol to indicate a variable, which can then be referenced later in your document.

```
$font-stack:   Helvetica, sans-serif
$primary-color: #333

body
  font: 100% $font-stack
  color: $primary-color
```

Sass 3 introduces a new syntax known as SCSS that is fully compatible with the syntax of CSS3, while still supporting the full power of Sass. It looks more like CSS.

```
$font-stack:   Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

The code above renders the following CSS.

```
body {
  font: 100% Helvetica, sans-serif;
  color: #333;
}
```

Another powerful feature of Sass is the ability to write nested CSS selectors. When writing HTML, you can often visualize the hierarchy of elements. Sass allows you to bring that hierarchy into your CSS.

```
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li {
    display: inline-block;
  }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```



Overly nested rules will result in overqualified CSS that can be hard to maintain.

As mentioned, Sass also supports partials, imports, mixins, and inheritance. Mixins can be particularly useful for handling vendor prefixes.

```
@mixin border-radius($radius) { ①
  -webkit-border-radius: $radius;
  -moz-border-radius: $radius;
  -ms-border-radius: $radius;
  border-radius: $radius;
}

.box { @include border-radius(10px); } ②
```

① Create a mixin using `@mixin` and give it a name. This uses `$radius` as a variable to set the radius value.

② Use `@include` followed by the name of the mixin.

CSS generated from the above code looks as follows.

```
.box {
  -webkit-border-radius: 10px;
  -moz-border-radius: 10px;
  -ms-border-radius: 10px;
  border-radius: 10px;
}
```

Bootstrap 3.x was written with [Less](#), a CSS pre-processor with similar features to Sass. It uses Sass for the 4.0+ versions.

JHipster allows you to use Sass by default. To learn more about structuring your CSS and naming classes, read the great [*Scalable and Modular Architecture for CSS*](#).

Webpack

JHipster 4+ uses [webpack](#) for building the client. JHipster 3.x used [Gulp](#). Gulp allows you to perform tasks like minification, concatenation, compilation (e.g. from TypeScript/CoffeeScript to JavaScript), unit testing, and more. webpack is a more modern solution that's become very popular for Angular projects and is included under the covers in Angular CLI.

webpack is a module bundler that recursively builds a dependency graph with every module your application needs. It packages all of these modules into a smaller number of bundles to be loaded by the browser. Its code-splitting abilities make it possible to break up large JavaScript applications into small chunks that can be loaded on demand.

It has four core concepts:

- **Entry:** This tells webpack where to start and follows the graph of dependencies to know what to bundle.
- **Output:** Once you've bundled all of your assets together, you need to tell webpack **where** to put them.
- **Loaders:** webpack treats every file (.css, .scss, .ts, .png, .html, etc.) as a module, but only understands JavaScript. Loaders transform files into modules as they are added to the dependency graph.
- **Plugins:** Loaders execute transforms per file. Plugins perform actions and customizations on chunks of your bundled modules.

Below is a basic `webpack.config.js` that shows all four concepts in use.

`webpack.config.js`

```

const HtmlWebpackPlugin = require('html-webpack-plugin'); //installed via npm
const webpack = require('webpack'); //to access built-in plugins
const path = require('path');

const config = {
  entry: './path/to/my/entry/file.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-first-webpack.bundle.js'
  },
  module: {
    rules: [
      { test: /\.txt$/, use: 'raw-loader' }
    ]
  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin(),
    new HtmlWebpackPlugin({template: './src/index.html'})
  ]
};

module.exports = config;

```

In the Angular section, I mentioned Angular CLI and my `ng-demo` project that shows how to use it. Angular CLI uses webpack internally, but you never see it because it wraps everything in its `ng` command. To see its webpack config, or to tweak it for your needs, you can *eject* it from your project by running `ng eject`.

I tried running this and found it added 20 dependencies to the project's `package.json` and generated a `webpack.config.js` file that's over 450 lines of code! This is roughly equivalent to the lines of code in a JHipster project's webpack configuration. JHipster generates a `webpack` directory that contains files for different scenarios: dev mode with hot reload, testing, and preparing for production.

- `logo-jhipster.png` is the logo that shows in desktop alerts for build notifications.
- `utils.js` contains utility functions for finding the project's version and determining external libraries.
- `webpack.common.js` is the common parent configuration that's extended by each of the following files.
- `webpack.dev.js` is the configuration used when you run `yarn start`. It enables hot-reloading with Browsersync and desktop notifications.
- `webpack.prod.js` is the configuration used for production. Angular's AOT compilation is used, HTML templates are converted to JavaScript, and source maps are created.

To learn more about webpack, I recommend visiting <https://webpack.academy>, which is a site dedicated to teaching webpack. It was created by [Sean Larkin](#), a member of the webpack core team, the Angular team, and the Angular CLI core team. He's a big reason for the success of webpack, as he's constantly promoting it, providing learning materials around it, and helping open-source projects adopt it.

WebSockets

WebSockets are an advanced technology that makes it possible to open an interactive communication channel between the user's browser and a server. With this API, you can send messages to a server and receive event-driven responses without having to poll the server for a reply. WebSockets have been called "TCP for the Web".

If you choose "WebSockets using Spring Websocket" as part of the "other technologies" options when creating a JHipster project, you'll get two JavaScript libraries added to your project:

- [webstomp-client](#): STOMP stands for “simple text-oriented messaging protocol”.
- [SockJS](#): SockJS provides a WebSocket-like object. If native WebSockets are not available, it falls back to other browser techniques.

To see how WebSockets work, take a look at the [JhiTrackerComponent](#) in a WebSockets-enabled project. This displays real-time activity information that's been posted to the [/websocket/tracker](#) endpoint.

src/main/webapp/app/admin/tracker/tracker.component.ts

```

import { Component, OnInit, OnDestroy } from '@angular/core';
import { JhiTrackerService } from 'app/core';

@Component({
  selector: 'jhi-tracker',
  templateUrl: './tracker.component.html'
})
export class JhiTrackerComponent implements OnInit, OnDestroy {
  activities: any[] = [];

  constructor(private trackerService: JhiTrackerService) {}

  showActivity(activity: any) {
    let existingActivity = false;
    for (let index = 0; index < this.activities.length; index++) {
      if (this.activities[index].sessionId === activity.sessionId) {
        existingActivity = true;
        if (activity.page === 'logout') {
          this.activities.splice(index, 1);
        } else {
          this.activities[index] = activity;
        }
      }
    }
    if (!existingActivity && activity.page !== 'logout') {
      this.activities.push(activity);
    }
  }

  ngOnInit() {
    this.trackerService.subscribe();
    this.trackerService.receive().subscribe(activity => {
      this.showActivity(activity);
    });
  }

  ngOnDestroy() {
    this.trackerService.unsubscribe();
  }
}

```

The **Tracker** service allows you to send tracking information — for example, to track when someone has authenticated.

src/main/webapp/app/core/tracker/tracker.service.ts

```

import { Injectable } from '@angular/core';
import { Router, NavigationEnd } from '@angular/router';
import { Observable, Observer, Subscription } from 'rxjs';

import { CSRFService } from '../auth/csrf.service';
import { WindowRef } from './window.service';
import { AuthServerProvider } from '../auth/auth-jwt.service';

import * as SockJS from 'sockjs-client';
import * as Stomp from 'webstomp-client';

@Injectable({ providedIn: 'root' })
export class JhiTrackerService {
    stompClient = null;
    subscriber = null;
    connection: Promise<any>;
    connectedPromise: any;
    listener: Observable<any>;
    listenerObserver: Observer<any>;
    alreadyConnectedOnce = false;
    private subscription: Subscription;

    constructor(
        private router: Router,
        private authServerProvider: AuthServerProvider,
        private $window: WindowRef,
        // tslint:disable-next-line: no-unused-variable
        private csrfService: CSRFService
    ) {
        this.connection = this.createConnection();
        this.listener = this.createListener();
    }

    connect() {
        if (this.connectedPromise === null) {
            this.connection = this.createConnection();
        }
        // building absolute path so that websocket doesn't fail when deploying with a context path
        const loc = this.$window.nativeWindow.location;
        let url;
        url = '// ' + loc.host + loc.pathname + 'websocket/tracker';
        const authToken = this.authServerProvider.getToken();
        if (authToken) {
            url += '?access_token=' + authToken;
        }
        const socket = new SockJS(url);
        this.stompClient = Stomp.over(socket);
        const headers = {};
    }
}

```

```

this.stompClient.connect(
    headers,
    () => {
        this.connectedPromise('success');
        this.connectedPromise = null;
        this.sendActivity();
        if (!this.alreadyConnectedOnce) {
            this.subscription = this.router.events.subscribe(event => {
                if (event instanceof NavigationEnd) {
                    this.sendActivity();
                }
            });
            this.alreadyConnectedOnce = true;
        }
    }
);

disconnect() {
    if (this.stompClient !== null) {
        this.stompClient.disconnect();
        this.stompClient = null;
    }
    if (this.subscription) {
        this.subscription.unsubscribe();
        this.subscription = null;
    }
    this.alreadyConnectedOnce = false;
}

receive() {
    return this.listener;
}

sendActivity() {
    if (this.stompClient !== null && this.stompClient.connected) {
        this.stompClient.send(
            '/topic/activity', // destination
            JSON.stringify({ page: this.router.routerState.snapshot.url }), // body
            {} // header
        );
    }
}

subscribe() {
    this.connection.then(() => {
        this.subscriber = this.stompClient.subscribe('/topic/tracker', data => {
            this.listenerObserver.next(JSON.parse(data.body));
        });
    });
}

```

```

unsubscribe() {
    if (this.subscriber !== null) {
        this.subscriber.unsubscribe();
    }
    this.listener = this.createListener();
}

private createListener(): Observable<any> {
    return new Observable(observer => {
        this.listenerObserver = observer;
    });
}

private createConnection(): Promise<any> {
    return new Promise((resolve, reject) => (this.connectedPromise = resolve));
}
}

```

WebSockets on the server side of a JHipster project are implemented with [Spring's WebSocket support](#). To learn more about WebSockets with Spring, see Baeldung's [Intro to WebSockets with Spring](#). The next section shows how a developer productivity tool that uses WebSockets implements something very cool.

Browsersync

[Browsersync](#) is one of those tools that makes you wonder how you ever lived without it. It keeps your assets in sync with your browser. It's also capable of syncing browsers, so you can, for example, scroll in Safari and watch synced windows scroll in Chrome and in Safari running in iOS Simulator. When you save files, it updates your browser windows, saving you an incredible amount of time. As its website says, "It's wicked-fast and totally free."

Browsersync is free to run and reuse, as guaranteed by its open-source Apache 2.0 License. It contains a number of slick features:

- Interaction sync: Browsersync mirrors your scroll, click, refresh, and form actions between browsers while you test.
- File sync: Browsers automatically update as you change HTML, CSS, images, and other project files.
- URL history: Browsersync records your test URLs so you can push them back out to all devices with a single click.
- Remote inspector: You can remotely tweak and debug web pages that are running on connected devices.

To integrate Browsersync in your project, you need a [package.json](#) and [gulpfile.js](#). Your [package.json](#) file only needs to contain a few things, weighing in at a slim 13 lines of JSON.

```
{
  "name": "jhipster-book",
  "version": "5.0.1",
  "description": "The JHipster Mini-Book",
  "repository": {
    "type": "git",
    "url": "git@bitbucket.org:mraible/jhipster-book.git"
  },
  "devDependencies": {
    "gulp": "3.9.1",
    "browser-sync": "2.24.7"
  }
}
```

The `gulpfile.js` utilizes the tools specified in `package.json` to enable Browsersync and create a magical web-development experience.

```
var gulp      = require('gulp'),
  browserSync = require('browser-sync').create();

gulp.task('serve', function() {

  browserSync.init({
    server: './src/main/webapp'
  });

  gulp.watch(['src/main/webapp/*.html', 'src/main/webapp/css/*.css'])
    .on('change', browserSync.reload);
});

gulp.task('default', ['serve']);
```

After you've created these files, you'll need to install [Node.js](#) and its package manager, npm. This should let you run the following command to install Browsersync and Gulp. You will only need to run this command when dependencies change in `package.json`.

```
npm install
```

Then run the following command to create a blissful development environment in which your browser auto-refreshes when files change on your hard drive.

gulp

JHipster integrates Browsersync for you, using webpack instead of Gulp. I show a Gulp example here because it so simple. I highly recommend Browsersync for your project. It's useful for determining if your web application can handle a page reload without losing the current user's state.

Summary

This section describes the UI components in a typical JHipster project. It taught you about the extremely popular UI framework called Angular. It showed you how to author HTML pages and use Bootstrap to make things look pretty. A build tool is essential for building a modern web application and it showed you how you can use webpack. Finally, it showed you how WebSockets work and described the beauty of Browsersync.

Now that you've learned about many of the UI components in a JHipster project, let's learn about the API side of things.

PART THREE

JHipster's API building blocks

JHipster is composed of two main components: a modern UI framework and an API. APIs are the modern data-retrieval mechanisms. Creating great UIs is how you make people smile.

Many APIs today are RESTful APIs. In fact, representational state transfer (REST) is the software architectural style of the World Wide Web. RESTful systems typically communicate over HTTP (Hypertext Transfer Protocol) using verbs (GET, POST, PUT, DELETE, etc.). This is the same way browsers retrieve web pages and send data to remote servers. REST was initially proposed by Roy Fielding in his 2000 Ph.D. dissertation, *Architectural Styles and the Design of Network-Based Software Architectures*.

JHipster leverages Spring MVC and its `@RestController` annotation to create a REST API. Its endpoints publish JSON to and consume JSON from clients. By separating the business logic and data persistence from the client, you can provide data to many different clients (HTML5, iOS, Android, TVs, watches, IoT devices, etc.). This also allows third-party and partner integration capabilities in your application. Spring Boot further complements Spring MVC by simplifying microservices and allowing you to create stand-alone JAR (Java Archive) files.

Spring Boot

In August 2013, Phil Webb and Dave Syer, engineers at Pivotal, [announced the first milestone release](#) of Spring Boot. Spring Boot makes it easy to create Spring applications with minimal effort. It takes an opinionated view of Spring and auto-configures dependencies for you. This allows you to write less code but still harness the power of Spring. The diagram below (from <https://spring.io>) shows how Spring Boot is the gateway to the larger Spring ecosystem.

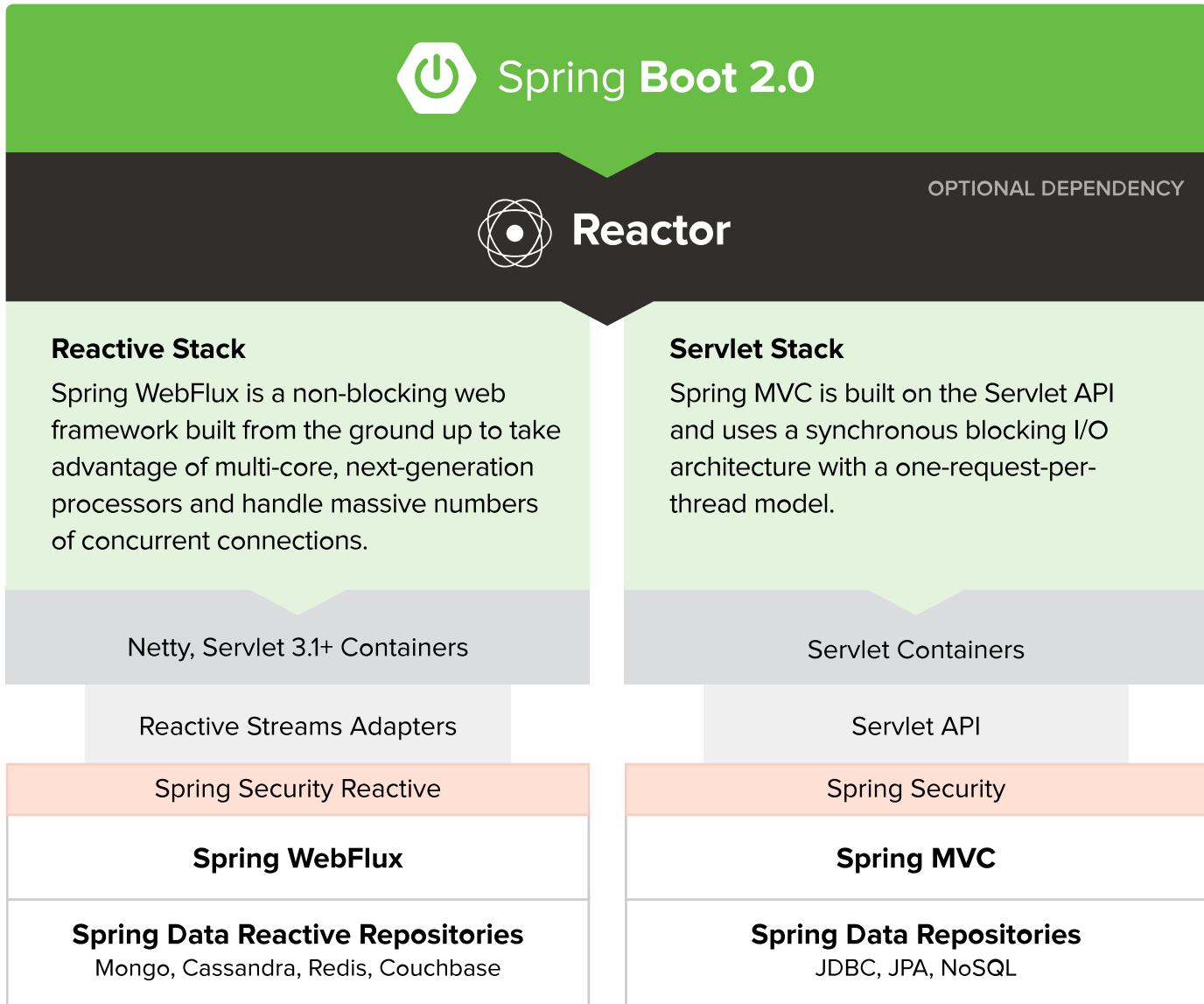


Figure 33. Spring Boot 2.0

The primary goals of Spring Boot are:

- to provide a radically faster and widely accessible “getting started” experience for all Spring development;
- to be opinionated out of the box, but get out of the way quickly as requirements start to diverge from the defaults; and
- to provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration).

Folks who want to use Spring Boot outside of a JHipster application can do so with Spring Initializr, a configurable service for generating Spring projects. You can visit it in your browser at <https://start.spring.io> or you can call it via `curl`.

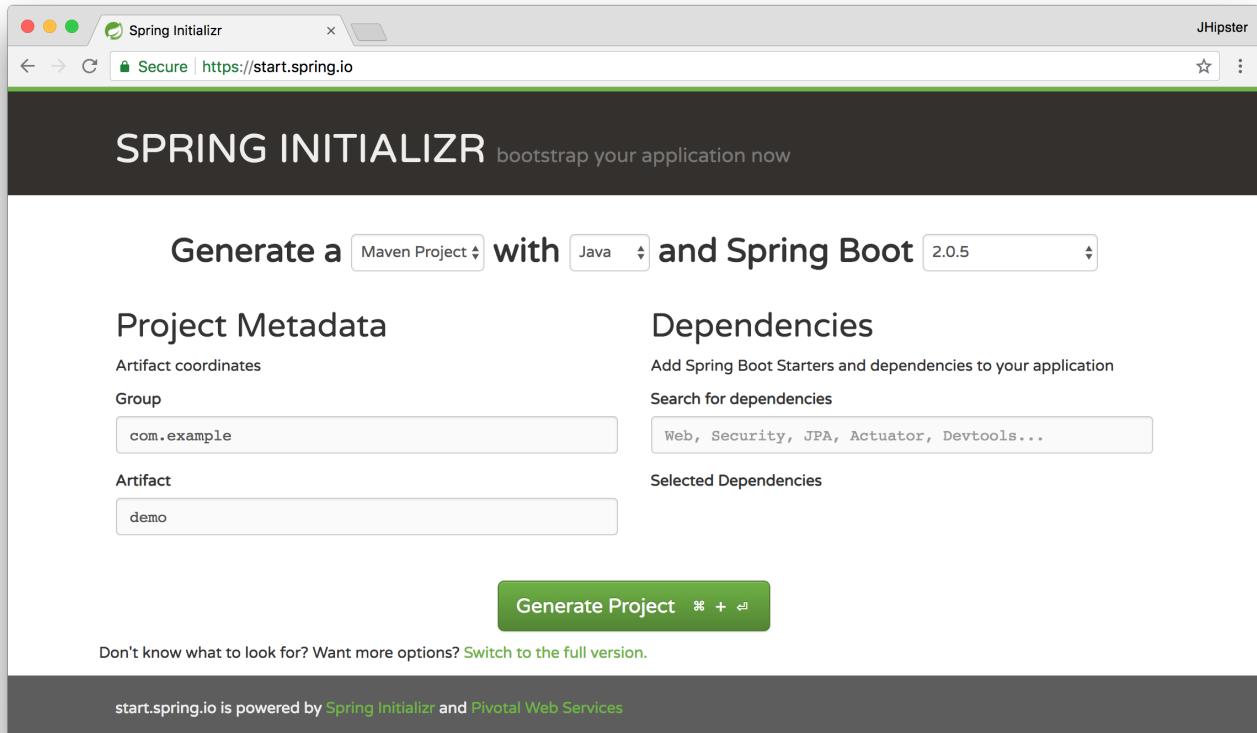


Figure 34. Spring Initializr in a browser

```
1. mraible@rogueone: ~ (zsh)
➜ ~ curl start.spring.io

.   _--'_--_(_)_--_--_ \\\ \
( ( )\__| '_| '_|| '_` | \\ \\
\W _--)| |_)| | | | || (| | ) ) )
'  |____| .__|_|_|_|_|_\_, | // //
=====|_|=====|_|/_=/_|/_/|
:: Spring Initializr ::  https://start.spring.io

This service generates quickstart projects that can be easily customized.
Possible customizations include a project's dependencies, Java version, and
build system or build structure. See below for further details.

The services uses a HAL based hypermedia format to expose a set of resources
to interact with. If you access this root resource requesting application/json
as media type the response will contain the following links:
+-----+-----+
| Rel | Description |
+-----+-----+
| gradle-build | Generate a Gradle build file |
| gradle-project | Generate a Gradle based project archive |
| maven-build | Generate a Maven pom.xml |
| maven-project * | Generate a Maven based project archive |
+-----+-----+
```

Figure 35. Spring Initializr via curl

Spring Initializr is an Apache 2.0-licensed open-source project that you install and customize to

generate Spring projects for your company or team. You can find it on GitHub at <https://github.com/spring-io/initializr>.

Spring Initializr is also available in the Eclipse-based Spring Tool Suite (STS) and IntelliJ IDEA.

Spring CLI

You can also download and install the Spring Boot CLI. The easiest way to install it is with [SDKMAN!](#)

```
curl -s "https://get.sdkman.io" | bash
sdk install springboot
```

Spring CLI is best used for rapid prototyping: when you want to show someone how to do something very quickly, with code you'll likely throw away when you're done. For example, if you want to create a “Hello World” web application in Groovy, you can do it with seven lines of code.

hello.groovy

```
@RestController
class WebApplication {
    @RequestMapping("/")
    String home() {
        "Hello World!"
    }
}
```

To compile and run this application, simply type:

```
spring run hello.groovy
```

After running this command, you can see the application at <http://localhost:8080>. For more information about the Spring Boot CLI, see [its documentation](#).

To show you how to create a simple application with Spring Boot, go to <https://start.spring.io> and select **Web**, **JPA**, **H2**, and **Actuator** as project dependencies. Click “Generate Project” to download a .zip file for your project. Extract it on your hard drive and import it into your favorite IDE.

This project has only a few files in it, as you can see by running the **tree** command (on *nix).

```

.
├── mvnw
├── mvnw.cmd
└── pom.xml
└── src
    ├── main
    │   ├── java
    │   │   └── com
    │   │       └── example
    │   │           └── demo
    │   │               └── DemoApplication.java
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        └── java
            └── com
                └── example
                    └── demo
                        └── DemoApplicationTests.java

```

14 directories, 6 files

`DemoApplication.java` is the heart of this application; the file and class name are not relevant. What is relevant is the `@SpringBootApplication` annotation and the class's `public static void main` method.

`src/main/java/com/example/demo/DemoApplication.java`

```

package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}

```

For this application, you'll create an entity, a JPA repository, and a REST endpoint to show data in the browser. To create an entity, add the following code to the `DemoApplication.java` file, outside of the `DemoApplication` class.

src/main/java/demo/com/example/demo/DemoApplication.java

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
...
@Entity
class Blog {

    @Id
    @GeneratedValue
    private Long id;
    private String name;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Blog{" +
            "id=" + id +
            ", name='" + name + '\'' +
            '}';
    }
}
```

In the same file, add a `BlogRepository` interface that extends `JpaRepository`. Spring Data JPA makes it really easy to create a CRUD repository for an entity. It automatically creates for you the implementation that talks to the underlying datastore.

src/main/java/com/example/demo/DemoApplication.java

```
import org.springframework.data.jpa.repository.JpaRepository;
...
interface BlogRepository extends JpaRepository<Blog, Long> {}
```

Define a `CommandLineRunner` that injects this repository and prints out all the data that's found by calling its `findAll()` method. `CommandLineRunner` is an interface that's used to indicate that a bean should run when it is contained within a `SpringApplication`.

src/main/java/com/example/demo/DemoApplication.java

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

...
@Component
class BlogCommandLineRunner implements CommandLineRunner {

    private BlogRepository repository;

    public BlogCommandLineRunner(BlogRepository repository) {
        this.repository = repository;
    }

    @Override
    public void run(String... strings) throws Exception {
        System.out.println(repository.findAll());
    }
}
```



Spring 4.3 added `implicit constructor injection`, eliminating the need for an `@Autowired` annotation.

To provide default data, create `src/main/resources/data.sql` and add a couple of SQL statements to insert data.

src/main/resources/data.sql

```
insert into blog (name) values ('First');
insert into blog (name) values ('Second');
```

Start your application with `mvn spring-boot:run` (or right-click → “Run in your IDE”) and you should see this default data show up in your logs.

```
2017-08-31 23:09:27.436 INFO 67327 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-08-31 23:09:27.470 INFO 67327 --- [main] o.h.h.i.QueryTranslatorFactoryInitiator : HHH000397: Using ASTQueryTranslatorFactory
[Blog{id=1, name='First'}, Blog{id=2, name='Second'}]
2017-08-31 23:09:27.549 INFO 67327 --- [main] com.example.demo.DemoApplication : Started DemoApplication in 3.924 seconds (JVM running for 4.492)
```

To publish this data as a REST API, create a `BlogController` class and add a `/blogs` endpoint that returns a list of blogs.

`src/main/java/demo/com/example/demo/DemoApplication.java`

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.Collection;
...
@RestController
class BlogController {
    private final BlogRepository repository;

    public BlogController(BlogRepository repository) {
        this.repository = repository;
    }

    @RequestMapping("/blogs")
    Collection<Blog> list() {
        return repository.findAll();
    }
}
```

After adding this code and restarting the application, you can `curl` the endpoint or open it in your favorite browser.

```
$ curl localhost:8080/blogs
[{"id":1,"name":"First"}, {"id":2,"name":"Second"}]
```



`HTTPie` is an alternative to cURL that makes many things easier.

Spring has one of the best track records for hipness in Javaland. It is an essential cornerstone of the solid API foundation that makes JHipster awesome. Spring Boot allows you to create stand-alone Spring applications that directly embed Tomcat, Jetty, or Undertow. It provides opinionated starter dependencies that simplify your build configuration, regardless of whether you're using Maven or Gradle.

External configuration

You can configure Spring Boot externally, 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 your configuration.

Spring Boot runs through this specific sequence for `PropertySource` to ensure that it overrides values sensibly:

1. Devtools global settings properties on your home directory (`~/.spring-boot-devtools.properties` when devtools is active).
2. `@TestPropertySource` annotations on your tests.
3. `@SpringBootTest#properties` annotation attribute on your tests.
4. Command-line arguments,
5. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
6. `ServletConfig` init parameters.
7. `ServletContext` init parameters.
8. JNDI attributes from `java:comp/env`.
9. Java System properties (`System.getProperties()`).
10. OS environment variables.
11. A `RandomValuePropertySource` that only has properties in `random.*`.
12. Profile-specific application properties outside of your packaged JAR (`application-{profile}.properties` and YAML variants).
13. Profile-specific application properties packaged inside your JAR (`application-{profile}.properties` and YAML variants).
14. Application properties outside of your packaged JAR (`application.properties` and YAML variants).
15. Application properties packaged inside your JAR (`application.properties` and YAML variants).
16. `@PropertySource` annotations on your `@Configuration` classes.
17. Default properties (specified using `SpringApplication.setDefaultProperties()`).

Application property files

By default, `SpringApplication` will load properties from `application.properties` files in the following locations and add them to the Spring `Environment`:

1. a `/config` subdirectory of the current directory,
2. the current directory,
3. a classpath `/config` package, and
4. the classpath root.



You can also use YAML (`.yml`) files as an alternative to properties files. JHipster uses YAML files for its configuration.

More information about Spring Boot's external-configuration feature can be found in Spring Boot's [“Externalized Configuration” reference documentation](#).

If you're using third-party libraries that require external configuration files, you may have issues loading them. These files might be loaded with:



`XXX.class.getResource().toURI().getPath()`

This code does not work when using a Spring Boot executable JAR because the classpath is relative to the JAR itself and not the filesystem. One workaround is to run your application as a WAR in a servlet container. You might also try contacting the maintainer of the third-party library to find a solution.

Automatic configuration

Spring Boot is unique in that it automatically configures Spring whenever possible. It does this by peeking into JAR files to see if they're hip. If they are, they contain a `META-INF/spring.factories` that defines configuration classes under the `EnableAutoConfiguration` key. For example, below is what's contained in `spring-boot-actuator-autoconfigure`.

`spring-boot-actuator-autoconfigure-2.0.5.RELEASE.jar!/META-INF/spring.factories`

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.actuate.autoconfigure.amqp.RabbitHealthIndicatorAutoConfiguration,\ 
org.springframework.boot.actuate.autoconfigure.audit.AuditAutoConfiguration,\ 
org.springframework.boot.actuate.autoconfigure.audit.AuditEventsEndpointAutoConfiguration,\ 
org.springframework.boot.actuate.autoconfigure.beans.BeansEndpointAutoConfiguration,\ 
org.springframework.boot.actuate.autoconfigure.cassandra.CassandraHealthIndicatorAutoConfiguration,\ 
org.springframework.boot.actuate.autoconfigure.cloudfoundry.servlet.CloudFoundryActuatorAutoConfiguration,\ 
org.springframework.boot.actuate.autoconfigure.cloudfoundry.reactive.ReactiveCloudFoundryActuatorAutoConfiguration,\ 
org.springframework.boot.actuate.autoconfigure.condition.ConditionsReportEndpointAutoConfiguration,\ 
org.springframework.boot.actuate.autoconfigure.context.properties.ConfigurationPropertiesReportEndpointAutoConfiguration,\ 
org.springframework.boot.actuate.autoconfigure.context.ShutdownEndpointAutoConfiguration,\ 
org.springframework.boot.actuate.autoconfigure.couchbase.CouchbaseHealthIndicatorAutoConfiguration,\
```

```

org.springframework.boot.actuate.autoconfigure.elasticsearch.ElasticsearchHealthIndicatorAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.endpoint.EndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.endpoint.jmx.JmxEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.endpoint.web.WebEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.env.EnvironmentEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.flyway.FlywayEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.health.HealthEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.health.HealthIndicatorAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.influx.InfluxDbHealthIndicatorAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.info.InfoContributorAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.info.InfoEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.jdbc.DataSourceHealthIndicatorAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.jms.JmsHealthIndicatorAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.jolokia.JolokiaEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.ldap.LdapHealthIndicatorAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.liquibase.LiquibaseEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.logging.LogFileWebEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.logging.LoggersEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.mail.MailHealthIndicatorAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.management.HeapDumpWebEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.management.ThreadDumpEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.CompositeMeterRegistryAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.MetricsAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.MetricsEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.amqp.RabbitMetricsAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.cache.CacheMetricsAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.export.atlas.AtlasMetricsExportAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.export.datadog.DatadogMetricsExportAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.export.ganglia.GangliaMetricsExportAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.export.graphite.GraphiteMetricsExportAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.export.influx.InfluxMetricsExportAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.export.jmx.JmxMetricsExportAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.export.newrelic.NewRelicMetricsExportAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.export.prometheus.PrometheusMetricsExportAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.export.simple.SimpleMetricsExportAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.export.signalfx.SignalFxMetricsExportAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.export.statsd.StatsdMetricsExportAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.export.wavefront.WavefrontMetricsExportAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.jdbc.DataSourcePoolMetricsAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.web.client.RestTemplateMetricsAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.web.reactive.WebFluxMetricsAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.web.servlet.WebMvcMetricsAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.metrics.web.tomcat.TomcatMetricsAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.mongo.MongoHealthIndicatorAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.neo4j.Neo4jHealthIndicatorAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.redis.RedisHealthIndicatorAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.scheduling.ScheduledTasksEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.session.SessionsEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.solr.SolrHealthIndicatorAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.system.DiskSpaceHealthIndicatorAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.trace.http.HttpTraceAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.trace.http.HttpTraceEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.web.mappings.MappingEndpointAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.web.reactive.ReactiveManagementContextAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.web.server.ManagementContextAutoConfiguration,\n
org.springframework.boot.actuate.autoconfigure.web.servlet.ServletManagementContextAutoConfiguration\n
org.springframework.boot.actuate.autoconfigure.web.ManagementContextConfiguration=\n
org.springframework.boot.actuate.autoconfigure.endpoint.web.ServletEndpointManagementContextConfiguration,\n
org.springframework.boot.actuate.autoconfigure.endpoint.web.reactive.WebFluxEndpointManagementContextConfiguration,\n
org.springframework.boot.actuate.autoconfigure.endpoint.web.servlet.WebMvcEndpointManagementContextConfiguration,\n

```

```
org.springframework.boot.actuate.autoconfigure.endpoint.web.jersey.JerseyWebEndpointManagementContextConfiguration,\norg.springframework.boot.actuate.autoconfigure.web.jersey.JerseyManagementChildContextConfiguration,\norg.springframework.boot.actuate.autoconfigure.web.reactive.ReactiveManagementChildContextConfiguration,\norg.springframework.boot.actuate.autoconfigure.web.servlet.ServletManagementChildContextConfiguration,\norg.springframework.boot.actuate.autoconfigure.web.servlet.WebMvcEndpointChildContextConfiguration\n\norg.springframework.boot.diagnostics.FailureAnalyzer=\norg.springframework.boot.actuate.autoconfigure.metrics.MissingRequiredConfigurationFailureAnalyzer
```

These configuration classes will usually contain `@Conditional` annotations to help configure themselves. Developers can use `@ConditionalOnMissingBean` to override the auto-configured defaults. There are several conditional-related annotations you can use when developing Spring Boot plugins:

- `@ConditionalOnClass` and `@ConditionalOnMissingClass`
- `@ConditionalOnMissingClass` and `@ConditionalOnMissingBean`
- `@ConditionalOnProperty`
- `@ConditionalOnResource`
- `@ConditionalOnWebApplication` and `@ConditionalOnNotWebApplication`
- `@ConditionalOnExpression`

These annotations are what give Spring Boot its immense power and make it easy to use, configure, and override.

Actuator

Spring Boot's Actuator sub-project adds several production-grade services to your application with little effort. You can add the actuator to a Maven-based project by adding the `spring-boot-starter-actuator` dependency.

```
<dependencies>\n    <dependency>\n        <groupId>org.springframework.boot</groupId>\n        <artifactId>spring-boot-starter-actuator</artifactId>\n    </dependency>\n</dependencies>
```

If you're using Gradle, you'll save a few lines:

```
dependencies {\n    compile("org.springframework.boot:spring-boot-starter-actuator")\n}
```

Actuator's main features are endpoints, metrics, auditing, and process monitoring. Actuator auto-creates a number of REST endpoints. By default, Spring Boot will also expose management endpoints

as JMX MBeans under the `org.springframework.boot` domain. Actuator REST endpoints include:

- `/auditevents` — Exposes audit events information for the current application.
- `/beans` — Returns a complete list of all the Spring beans in your application.
- `/conditions` — Shows the conditions that were evaluated on configuration and auto-configuration classes.
- `/configprops` — Returns a list of all `@ConfigurationProperties`.
- `/env` — Returns properties from Spring's `ConfigurableEnvironment`.
- `/flyway` — Shows any Flyway database migrations that have been applied.
- `/health` — Returns information about application health.
- `/httptrace` — Returns trace information (by default, the last 100 HTTP requests).
- `/info` — Returns basic application info.
- `/loggers` — Shows and modifies the configuration of loggers in the application.
- `/liquibase` — Shows any Liquibase database migrations that have been applied.
- `/metrics` — Returns performance information for the current application.
- `/mappings` — Returns a list of all `@RequestMapping` paths.
- `/scheduledtasks` — Displays the scheduled tasks in your application.
- `/sessions` — Allows retrieval and deletion of user sessions from a Spring Session-backed session store.
- `/shutdown` — Shuts the application down gracefully (not enabled by default).
- `/threaddump` — Performs a thread dump.

JHipster includes a plethora of Spring Boot starter dependencies by default. This allows developers to write less code and worry less about dependencies and configuration. The boot-starter dependencies in the 21-Points Health application are as follows:

```
spring-boot-starter-cache
spring-boot-starter-mail
spring-boot-starter-logging
spring-boot-starter-actuator
spring-boot-starter-aop
spring-boot-starter-data-jpa
spring-boot-starter-data-elasticsearch
spring-boot-starter-data-jest
spring-boot-starter-security
spring-boot-starter-web
spring-boot-starter-undertow
spring-boot-starter-thymeleaf
spring-boot-starter-test
```

Spring Boot does a great job of auto-configuring libraries and simplifying Spring. JHipster complements that by integrating the wonderful world of Spring Boot with a modern UI and developer experience.

Spring WebFlux

Spring Boot 2.0 also supports building applications with a reactive stack through Spring WebFlux. When using WebFlux (instead of Web), your application will be based on the Reactive Streams API and run on non-blocking servers such as Netty, Undertow, and Servlet 3.1+ containers.

At the time of this writing, JHipster has experimental support for generating microservice apps with WebFlux. See [pull request #7983](#) for more information.

Showing how Spring WebFlux works is outside the scope of this mini-book. If you'd like to learn more about it, I'd suggest you read Josh Long and my [Build Reactive APIs with Spring WebFlux](#) blog post.

Maven versus Gradle

Maven and Gradle are the two main build tools used in Java projects today. JHipster allows you to use either one. With Maven, you have one `pom.xml` file that's 1090 lines of XML. With Gradle, you end up with several `*.gradle` files. In the 21-Points project, the Groovy code adds up to only 496 lines.

Apache calls [Apache Maven](#) a “software project-management and comprehension tool”. Based on the concept of a project object model (POM), Maven can manage a project’s build, reporting, and documentation from a central piece of information. Most of Maven’s functionality comes through plugins. There are Maven plugins for building, testing, source-control management, running a web server, generating IDE project files, and much more.

[Gradle](#) is a general-purpose build tool. It can build pretty much anything you care to implement in your build script. Out of the box, however, it won’t build anything unless you add code to your build

script to ask for that. Gradle has a Groovy-based domain-specific language (DSL) instead of the more traditional XML form of declaring the project configuration. Like Maven, Gradle has plugins that allow you to configure tasks for your project. Most plugins add some preconfigured tasks, which together do something useful. For example, Gradle's Java plugin adds tasks to your project that will compile and unit test your Java source code as well as bundle it into a JAR file.

In January 2014, ZeroTurnaround's RebelLabs published a report titled [Java Build Tools – Part 2: A Decision Maker's Comparison of Maven, Gradle and Ant + Ivy](#), which provided a timeline of build tools from 1977 through 2013.

THE EVOLUTION OF BUILD TOOLS: 1977 - 2013 (AND BEYOND)

Visual timeline

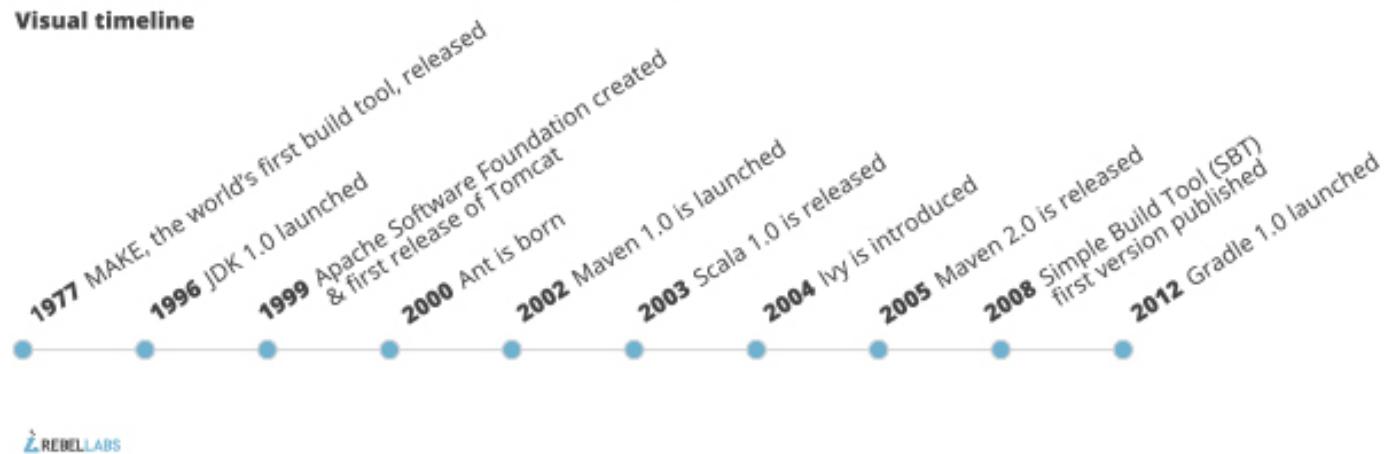


Figure 36. The Evolution of Build Tools, 1977-2013

Back then, RebelLabs advised that you experiment with Gradle in your next project.

If we were forced to conclude with any general recommendation, it would be to go with Gradle if you are starting a new project.

— RebelLabs, Java Build Tools – Part 2: A Decision Maker's Comparison of Maven, Gradle and Ant + Ivy

I've used both tools for building projects and they've both worked quite well. Maven works for me, but I've used it for over 10 years and recognize that my history and experience with it might contribute to my bias towards it. If you prefer Gradle simply because you are trying to avoid XML, [Polyglot for Maven](#) may change your perspective. It supports Atom, Groovy, Clojure, Ruby, Scala, and YAML languages. Ironically, you need to include a XML file to use it. To add support for non-XML languages, create a `#{project}/.mvn/extensions.xml` file and add the following XML to it.

```
<?xml version="1.0" encoding="UTF-8"?>
<extensions>
  <extension>
    <groupId>io.takari.polyglot</groupId>
    <artifactId>${artifactId}</artifactId>
    <version>0.3.0</version>
  </extension>
</extensions>
```

In this example, `${artifactId}` should be `polyglot-language`, where `language` is one of the aforementioned languages.

To convert an existing `pom.xml` file to another format, you can use the following command.

```
mvn io.takari.polyglot:polyglot-translate-plugin:translate \
-Dinput=pom.xml -Doutput=pom.{format}
```

Supported formats are `rb`, `groovy`, `scala`, `yaml`, `atom`, and of course `xml`. You can even convert back to XML or cross-convert between all supported formats. To learn more about alternate languages with Maven, see [Polyglot for Maven](#) on GitHub.

Many Internet resources support the use of Gradle. There's Gradle's own [Gradle vs Maven Feature Comparison](#). Benjamin Muschko, a principal engineer at Gradle, wrote a Dr. Dobb's article titled "[Why Build Your Java Projects with Gradle Rather than Ant or Maven?](#)" He's also the author of [Gradle in Action](#).

Gradle is the default build tool for Android development. Android Studio uses a Gradle wrapper to fully integrate the Android plugin for Gradle.



Both Maven and Gradle provide wrappers that allow you to embed the build tool within your project and source-control system. This allows developers to build or run the project after only installing Java. Since the build tool is embedded, they can type `gradlew` or `mvnw` to use the embedded build tool.

Regardless of which you prefer, Spring Boot supports both Maven and Gradle. You can learn more by visiting their respective documentation pages:

- [Spring Boot Maven plugin](#)
- [Spring Boot Gradle plugin](#)

I'd recommend starting with the tool that's most familiar to you. If you're using JHipster for the first time, you'll want to limit the number of new technologies you have to deal with. You can always add some for your next application. JHipster is a great learning tool, and you can also generate your project

with a different build tool to see what that looks like.

IDE support: Running, debugging, and profiling

IDE stands for “integrated development environment”. It is the lifeblood of a programmer who likes keyboard shortcuts and typing fast. The good IDEs have code completion that allows you to type a few characters, press tab, and have your code written for you. Furthermore, they provide quick formatting, easy access to documentation, and debugging. You can generate a lot of code with your IDE in statically typed languages like Java, like getters and setters on POJOs and methods in interfaces and classes. You can also easily find references to methods.

The JHipster documentation includes [guides](#) for configuring Eclipse, IntelliJ IDEA, Visual Studio Code, and NetBeans. Not only that, but Spring Boot has a devtools plugin that’s configured by default in a generated JHipster application. This plugin allows hot-reloading of your application when you recompile classes.

[IntelliJ IDEA](#), which brings these same features to Java development, is a truly amazing IDE. If you’re only writing JavaScript, their [WebStorm IDE](#) will likely become your best friend. Both IntelliJ products have excellent CSS support and accept plugins for many web languages/frameworks. To make IDEA auto-compile on save, like Eclipse does, perform the following steps:

- Navigate to File > Settings > Build, Execution, Deployment > Compiler: enable [Make project automatically](#)
- Open Registry (Mac: [Cmd+Shift+A](#), Linux: [Ctrl+Shift+A](#), choose Registry...) and enable [compiler.automake.allow.when.app.running](#)

[Eclipse](#) is a free alternative to IntelliJ IDEA. Its error highlighting (via auto-compile), code assist, and refactoring support is excellent. When I started using it back in 2002, it blew away the competition. It was the first Java IDE that was fast and efficient to use. Unfortunately, it fell behind in the JavaScript MVC era and lacks good support for JavaScript or CSS.

NetBeans has a [Spring Boot plugin](#). The NetBeans team has been doing a lot of work on web-tools support; they have good JavaScript/AngularJS support and there’s a [NetBeans Connector](#) plugin for Chrome that allows two-way editing in NetBeans and Chrome.

[Visual Studio Code](#) is an open-source text editor made by Microsoft. It’s become a popular editor for TypeScript and has plugins for Java development.

The beauty of Spring Boot is you can run it as a simple Java process. This means you can right-click on your [*Application.java](#) class and run it (or debug it) from your IDE. When debugging, you’ll be able to set breakpoints in your Java classes and see what variables are being set to before a process executes.

To learn about profiling a Java application, I recommend you watch Nitsan Wakart’s [“Java Profiling from the Ground Up!”](#) To learn more about memory and JavaScript applications, I recommend Addy Osmani’s [“JavaScript Memory Management Masterclass”](#).

Security

Spring Boot has excellent security features thanks to its integration with Spring Security. When you create a Spring Boot application with a [spring-boot-starter-security](#) dependency, you get HTTP Basic authentication out of the box. By default, a user is created with username `user` and the password is printed in the logs when the application starts. To override the generated password, you can define a `spring.security.user.password`. Additional security features of Spring Boot can be found in [Spring Boot's guide to security](#).

The most basic Spring Security Java configuration creates a servlet [Filter](#), which is responsible for all the security (protecting URLs, validating credentials, redirecting to login, etc.). This involves several lines of code, but half of them are class imports.

```
import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

import static org.springframework.security.core.userdetails.User.UserBuilder;

@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    public UserDetailsService userDetailsService() {
        // ensure the passwords are encoded properly
        UserBuilder users = User.withDefaultPasswordEncoder();
        InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();
        manager.createUser(users.username("user").password("password").roles("USER").build());
        return manager;
    }
}
```

There's not much code, but it provides many features:

- It requires authentication to every URL in your application.
- It generates a login form for you.
- It allows user:password to authenticate with form-based authentication.
- It allows the user to log out.
- It prevents CSRF attacks.
- It protects against session fixation.
- It includes security-header integration with:

- HTTP Strict Transport Security for secure requests,
- X-Content-Type-Options integration,
- cache control,
- X-XSS-Protection integration, and
- X-Frame-Options integration to help prevent clickjacking.
- It integrates with HttpServletRequest API methods of: `getRemoteUser()`, `getUserPrincipal()`, `isUserInRole(role)`, `login(username, password)`, and `logout()`.

JHipster takes the excellence of Spring Security and uses it to provide the real-world authentication mechanism that applications need. When you create a new JHipster project, it provides you with three authentication options:

- **JWT authentication** — A stateless security mechanism. JSON Web Token (JWT) is an [IETF proposed standard](#) that uses a compact, URL-safe means of representing claims to be transferred between two parties. JHipster's implementation uses the [Java JWT project](#).
- **HTTP Session Authentication** — Uses the HTTP session, so it is a stateful mechanism. Recommended for small applications.
- **OAuth 2.0 / OIDC Authentication** — A stateful security mechanism, like HTTP Session. You might prefer it if you want to share your users between several applications.

OAuth 2.0

[OAuth 2.0](#) is the current version of the OAuth framework (originally created in 2006). OAuth 2.0 focuses on simplifying client development while supporting web applications, desktop applications, mobile phones, and living-room devices. If you'd like to learn about how OAuth works, see [What the Heck is OAuth?](#)

In addition to authentication choices, JHipster offers security improvements: improved “remember me” (unique tokens stored in database), cookie-theft protection, and CSRF protection.

By default, JHipster comes with four different users:

- **system** — Used by audit logs when something is done automatically.
- **anonymousUser** — Anonymous users when they do an action.
- **user** — A normal user with “ROLE_USER” authorization; the default password is “user”.
- **admin** — An admin user with “ROLE_USER” and “ROLE_ADMIN” authorizations; the default password is “admin”.

For security reasons, you should change the default passwords in `src/main/resources/config/liquibase/users.csv` or through the User Management feature when

deployed.

JPA versus MongoDB versus Cassandra

A traditional relational-database management system (RDBMS) provides a number of properties that guarantee its transactions are processed reliably: ACID, for atomicity, consistency, isolation, and durability. Databases like MySQL and PostgreSQL provide RDBMS support and have done wonders to reduce the costs of databases. JHipster supports vendors like Oracle and Microsoft as well. If you'd like to use a traditional database, select SQL when creating your JHipster project.



JHipster's [Using Oracle](#) guide explains how you need an Oracle account to download its proprietary JDBC driver.

NoSQL databases have helped many web-scale companies achieve high scalability through [eventual consistency](#): because a NoSQL database is often distributed across several machines, with some latency, it guarantees only that all instances will eventually be consistent. Eventually consistent services are often called BASE (basically available, soft state, eventual consistency) services in contrast to traditional ACID properties.

When you create a new JHipster project, you'll be prompted with the following.

```
? Which *type* of database would you like to use? (Use arrow keys)
  SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
  MongoDB
  Couchbase
  Cassandra
```

If you're familiar with RDBMS databases, I recommend you use PostgreSQL or MySQL for both development and production. PostgreSQL has great support on Heroku and MySQL has excellent support on AWS. JHipster's [AWS sub-generator](#) has a limitation of only working with MySQL.

If your idea is the next Facebook, you might want to consider a NoSQL database that's more concerned with performance than third normal form.

NoSQL encompasses a wide variety of different database technologies that were developed in response to a rise in the volume of data stored about users, objects, and products, the frequency in which this data is accessed, and performance and processing needs. Relational databases, on the other hand, were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the cheap storage and processing power available today.

— MongoDB, [NOSQL Database Explained](#)

MongoDB was founded in 2007 by the folks behind DoubleClick, ShopWiki, and Gilt Groupe. It uses the Apache and GNU-APGL licenses on [GitHub](#). Its many large customers include Adobe, eBay, and eHarmony.

[Cassandra](#) is “a distributed storage system for managing structured data that is designed to scale to a very large size across many commodity servers, with no single point of failure” (from “[Cassandra – A structured storage system on a P2P Network](#)” on the Facebook Engineering blog). It was initially developed at Facebook to power its Inbox Search feature. Its creators, Avinash Lakshman (one of the creators of Amazon DynamoDB) and Prashant Malik, released it as an open-source project in July 2008. In March 2009, it became an Apache Incubator project, and graduated to a top-level project in February 2010.

In addition to Facebook, Cassandra helps a number of other companies achieve web scale. It has some impressive numbers about scalability on its homepage.

One of the largest production deployments is Apple’s, with over 75,000 nodes storing over 10 PB of data. Other large Cassandra installations include Netflix (2,500 nodes, 420 TB, over 1 trillion requests per day), Chinese search engine Easou (270 nodes, 300 TB, over 800 million requests per day), and eBay (over 100 nodes, 250 TB).

— Cassandra, [Project Homepage](#)

JHipster’s data support lets you dream big!

NoSQL with JHipster

When MongoDB is selected:

- JHipster will use Spring Data MongoDB, similar to Spring Data JPA.
- JHipster will use [Mongobee](#) instead of Liquibase to manage database migrations.
- The entity sub-generator will not ask you about relationships. You can't have relationships with a NoSQL database.
- `de.flapdoodle.embed.mongo` is used to run an in-memory version of the database for running unit tests.

Liquibase

[Liquibase](#) is “source control for your database”. It’s an open-source (Apache 2.0) project that allows you to manipulate your database as part of a build or runtime process. It allows you to diff your entities against your database tables and create migration scripts. It even allows you to provide comma-delimited default data! For example, default users are loaded from `src/main/resources/config/liquibase/users.csv`.

This file is loaded by Liquibase when it creates the database schema.

`src/main/resources/config/liquibase/changelog/0000000000000000_initial_schema.xml`

```
<loadData encoding="UTF-8"
    file="config/liquibase/users.csv"
    separator=";"
    tableName="jhi_user">
    <column name="activated" type="boolean"/>
    <column name="created_date" type="timestamp"/>
</loadData>
<dropDefaultValue tableName="jhi_user" columnName="created_date" columnDataType="datetime"/>
```

Liquibase supports [most major databases](#). If you use MySQL or PostgreSQL, you can use `mvn liquibase:diff` (or `./gradlew generateChangeLog`) to automatically generate a changelog.

[JHipster’s development guide](#) recommends the following workflow:

1. Modify your JPA entity (add a field, a relationship, etc.).
2. Run `mvn compile liquibase:diff`.
3. A new changelog is created in your `src/main/resources/config/liquibase/changelog` directory.
4. Review this changelog and add it to your `src/main/resources/config/liquibase/master.xml` file, so it

is applied the next time you run your application.

If you use Gradle, you can use the same workflow by running `./gradlew generateChangeLog`.

Elasticsearch

Elasticsearch adds searchability to your entities. JHipster's Elasticsearch support requires using a SQL database. Spring Boot uses and configures [Spring Data Elasticsearch](#). When using JHipster's entity sub-generator, it automatically indexes the entity and creates an endpoint to support searching its properties. Search superpowers are also added to the Angular UI, so you can search in your entity's list screen.

When using the (default) “dev” profile, the in-memory Elasticsearch instance will store files in the `build/elasticsearch` folder. You can change this by modifying the following setting in `application-dev.yml`.

`src/main/resources/config/application-dev.yml`

```
data:
  elasticsearch:
    properties:
      path:
        home: build/elasticsearch
```

When using the “prod” profile, JHipster will use Spring Data Jest to communicate with Elasticsearch’s REST API on port `9200`. This setting is configured in `application-prod.yml`.

`src/main/resources/config/application-prod.yml`

```
data:
  jest:
    uri: http://localhost:9200
```

If you want to run the “prod” profile locally, you will need to start an Elasticsearch Docker image first.

```
docker-compose -f src/main/docker/elasticsearch.yml up -d
```

Elasticsearch is used by a number of well-known companies: Facebook, GitHub, and Uber among others. The project is backed by [Elastic](#), which provides an ecosystem of projects around Elasticsearch. Some examples are:

- [Elasticsearch as a Service](#) — “Hosted and managed Elasticsearch”.
- [Logstash](#) — “Process any data, from any source”.

- [Kibana](#) — “Explore and visualize your data”.

The ELK (Elasticsearch, Logstash, and Kibana) stack is all open-source projects sponsored by Elastic. It's a powerful solution for monitoring your applications and seeing how they're being used.

Deployment

A JHipster application can be deployed wherever a Java program can be run. Spring Boot uses a `public static void main` entry point that launches an embedded web server for you. Spring Boot applications are embedded in a “fat JAR”, which includes all necessary dependencies like, for example, the web server and start/stop scripts. You can give anybody this `.jar` and they can easily run your app: no build tool required, no setup, no web-server configuration, etc. It's just `java -jar killerapp.jar`.



Josh Long's [“Deploying Spring Boot Applications”](#) is an excellent resource for learning how to customize your application archive. It shows how to change your application to a traditional WAR: extend `SpringBootServletInitializer`, change packaging to `war`, and set `spring-boot-starter-tomcat` as a provided dependency.

To build your JHipster app with the production profile, use the preconfigured “prod” Maven profile.

```
mvn -Pprod package
```

With Gradle, it's:

```
gradlew -Pprod bootWar
```

The “prod” profile will trigger a `webpack:prod`, which optimizes your static resources. It will combine your JavaScript and CSS files, minify them, and get them production ready. It also updates your HTML (in your `(build|target)/www` directory) to have references to your versioned, combined, and minified files.

A JHipster application can be deployed to your own JVM, [Cloud Foundry](#), [Heroku](#), [Kubernetes](#), and [AWS](#).

I've deployed JHipster applications to Heroku, Cloud Foundry, and Google Cloud with Kubernetes.

Summary

The Spring Framework has one of the best track records for hipness in Javaland. It's remained backwards compatible between many releases and has lived as an open-source project for more than 14 years. Spring Boot has provided a breath of fresh air for people using Spring with its starter dependencies, auto-configuration, and monitoring tools. It's made it easy to build microservices on the

JVM and deploy them to the cloud.

You've seen some of the cool features of Spring Boot and the build tools you can use to package and run a JHipster application. I've described the power of Spring Security and showed you its many features, which you can enable with only a few lines of code. JHipster supports both relational databases and NoSQL databases, which allows you to choose how you want your data stored. You can choose JPA, MongoDB, or Cassandra when creating a new application.

Liquibase will create your database schema for you and help you update your database when the need arises. It provides an easy-to-use workflow to adding new properties to your JHipster-generated entities using its diff feature.

You can add rich search capabilities to your JHipster app with Elasticsearch. This is one of the most popular Java projects on GitHub and there's a reason for that: it works really well.

JHipster applications are Spring Boot applications, so you can deploy them wherever Java can be run. You can deploy them in a traditional Java EE (or servlet) container or you can deploy them in the cloud. The sky's the limit!

PART FOUR

Microservices with JHipster

Adopting a microservice architecture provides unique opportunities to add failover and resiliency to your systems, so your components can handle load spikes and errors gracefully. Microservices make change less expensive, too. They can also be a good idea when you have a large team working on a single product. You can likely break up your project into components that can function independently of one another. Once components can function independently, they can be built, tested, and deployed independently. This gives an organization and its teams the agility to develop and deploy very quickly.

Before we dive into the code tutorial, I'd like to talk a bit about microservices, their history, and why you should (or should not) consider a microservices architecture for your next project.

History of microservices

According to [Wikipedia](#), the term "microservice" was first used as a common architecture style at a workshop of software architects near Venice in May 2011. In May 2012, the same group decided "microservices" was a more appropriate name.

[Adrian Cockcroft](#), who was at Netflix at the time, described this architecture as "fine-grained SOA". Martin Fowler and James Lewis wrote an article titled simply "[Microservices](#)" on March 25, 2014. Years later, this is still considered the definitive article for microservices.

Organizations and Conway's law

Technology has traditionally been organized into technology layers: UI team, database team, operations team. When teams are separated along these lines, even simple changes can lead to a cross-team project consuming huge chunks of time and budget.

A smart team will optimize around this and choose the lesser of two evils: forcing the logic into whichever application they have access to. This is an example of [Conway's law](#) in action.

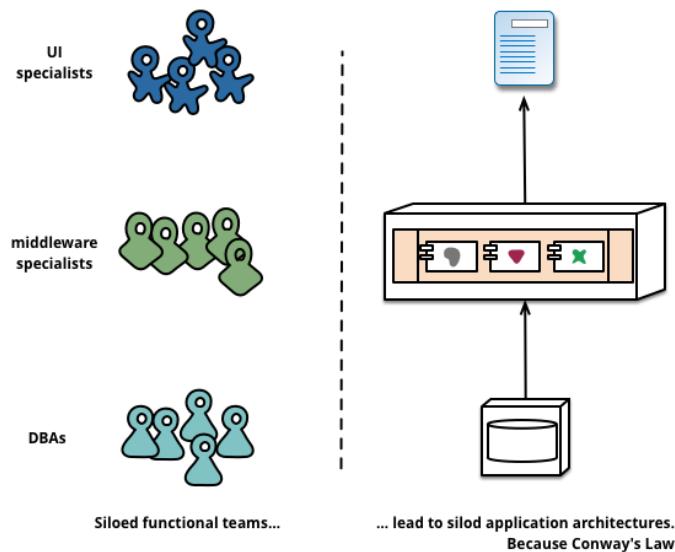


Figure 37. Conway's law

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

— Melvyn Conway, 1967

Microservices architecture philosophy

The philosophy of a microservices architecture is essentially equal to the Unix philosophy of "do one thing and do it well". The characteristics of a microservices architecture are as follows:

- componentization via services,
- organized around business capabilities,
- products not projects,
- smart endpoints and dumb pipes,
- decentralized governance,
- decentralized data management,
- infrastructure automation,
- design for failure, and
- evolutionary design.

Why microservices?

For most developers, dev teams, and organizations, it's easier to work on small, "do one thing well" services. No single program represents the whole application, so services can change frameworks (or even languages) without a massive cost. As long as the services use a language-agnostic protocol (HTTP or lightweight messaging), applications can be written in several different platforms — Java, Ruby, Node, Go, .NET, etc. — without issue.

Platform-as-a-service (PaaS) providers and containers have made it easy to deploy microservices. All the technologies needed to support a monolith (e.g., load balancing, discovery, process monitoring) are provided by the PaaS, outside of your container. Deployment effort comes close to zero.

Are microservices the future?

The consequences of architecture decisions, like adopting microservices, usually only become evident several years after you make them. Microservices have been successful at companies like LinkedIn, Twitter, Facebook, Amazon.com, and Netflix, but that doesn't mean they'll be successful for your organization. Component boundaries are hard to define. If you're not able to create your components cleanly, you're just shifting complexity from inside a component to the connections between components. Also, team capabilities are something to consider. A weak team will always create a weak

system.

You shouldn't start with a microservices architecture. Instead, begin with a monolith, keep it modular, and split it into microservices once the monolith becomes a problem.

— Martin Fowler

Microservices with JHipster

In this example, I'll show you how to build a microservices architecture with JHipster. As part of this process, you'll be generating three applications and running several others.

- Generate a gateway.
- Generate a blog microservice.
- Generate a store microservice.
- Run the JHipster Registry, Keycloak, and MongoDB.

You can see how these components fit in the diagram below.

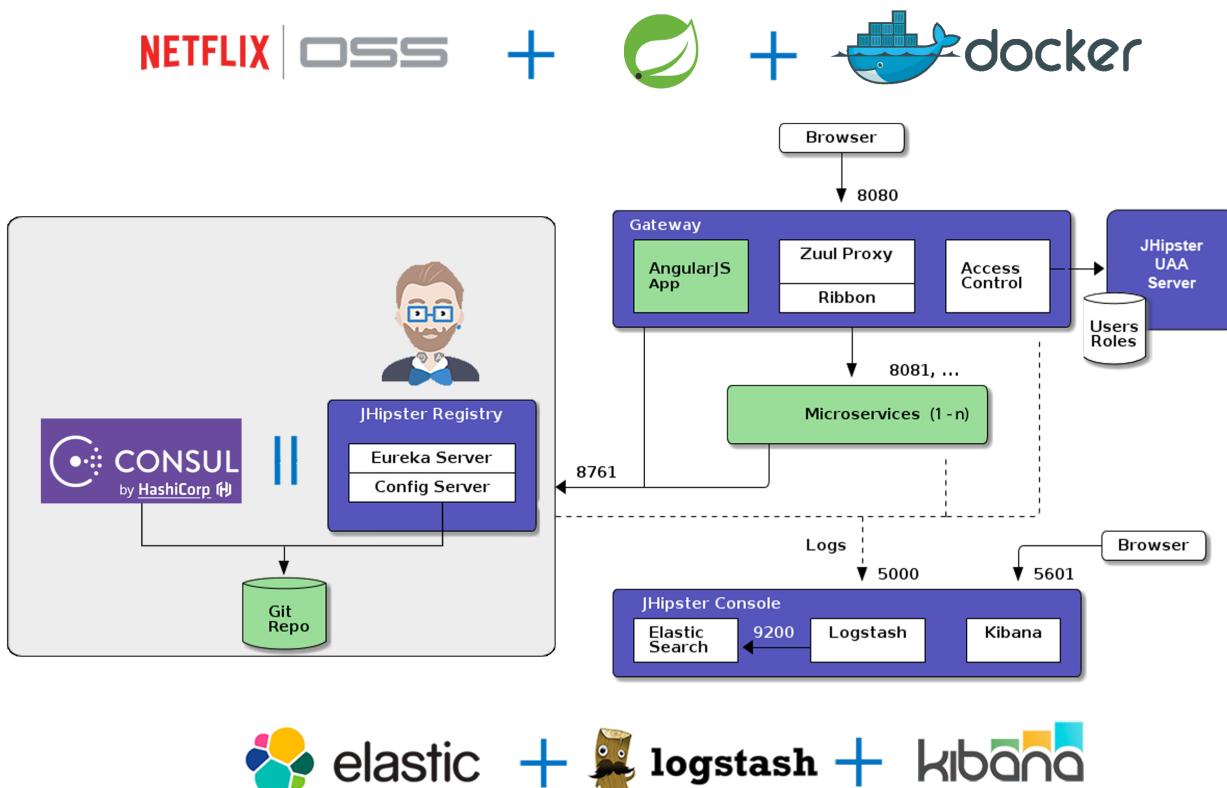


Figure 38. JHipster microservices architecture

To see what's happening inside your applications, you can use the [JHipster Console](#), a monitoring tool based on the [Elastic Stack](#). I'll cover JHipster Console in the Docker Compose section.

This tutorial shows you how to build a microservices architecture with JHipster 5.4.2. You'll generate a gateway (powered by Netflix Zuul), a blog microservice (that talks to PostgreSQL), a store microservice (that uses MongoDB), and use Docker Compose to make sure it all runs locally. Then you'll deploy it all to Heroku.

Generate an API gateway and microservice applications

One of the new features added in JHipster 5.3.x is the ability to generate a full microservices stack using the `import-jdl` command. Open a terminal window, create a directory (e.g., `jhipster-microservices-example`) and create an `apps.jh` file in it. Copy the JDL below into this file.

Example 1. apps.jh

```
application { ①
  config {
    baseName gateway,
    packageName com.okta.developer.gateway,
    applicationType gateway,
    authenticationType oauth2, ②
    prodDatabaseType postgresql,
    searchEngine elasticsearch, ③
    serviceDiscoveryType eureka, ④
    testFrameworks [protractor] ⑤
  }
  entities Blog, Post, Tag, Product
}

application {
  config {
    baseName blog,
    packageName com.okta.developer.blog,
    applicationType microservice, ⑥
    authenticationType oauth2, ⑦
    prodDatabaseType postgresql,
    searchEngine elasticsearch,
    serverPort 8081, ⑧
    serviceDiscoveryType eureka
  }
  entities Blog, Post, Tag
}

application {
```

```

config {
    baseName store,
    packageName com.okta.developer.store,
    applicationType microservice,
    authenticationType oauth2,
    databaseType mongodb, ⑨
    devDatabaseType mongodb,
    prodDatabaseType mongodb,
    enableHibernateCache false,
    searchEngine elasticsearch,
    serverPort 8082,
    serviceDiscoveryType eureka
}
entities Product
}

⑩
entity Blog {
    name String required minlength(3),
    handle String required minlength(2)
}

entity Post {
    title String required,
    content TextBlob required,
    date Instant required
}

entity Tag {
    name String required minlength(2)
}

entity Product {
    title String required,
    price BigDecimal required min(0),
    image ImageBlob
}

⑪
relationship ManyToOne {
    Blog{user(login)} to User,
    Post{blog(name)} to Blog
}

relationship ManyToMany {
    Post{tag(name)} to Tag{post}
}

```

⑫

paginate Post, Tag with infinite-scroll
 paginate Product with pagination

⑬

microservice Product with store
 microservice Blog, Post, Tag with blog

- ① JDL supports full stack definitions in v3.0!
- ② The authentication type for the gateway is OAuth 2.0.
- ③ Specifying `elasticsearch` on the gateway is necessary for JHipster to generate the proper client code for microservices that use Elasticsearch.
- ④ You must specify `eureka` as the service discovery type for the gateway and all microservice apps.
- ⑤ Including Protractor on the gateway allows you to test everything with `npm run e2e`.
- ⑥ For the microservice apps, you need to specify an application type of `microservice`.
- ⑦ The microservice app's authentication type must match the gateway.
- ⑧ The default server port is 8080. You must specify different ports for each app.
- ⑨ The store app uses MongoDB for its database. You must use the same databases for dev and prod when using NoSQL options.
- ⑩ Entity definitions live outside of your application definitions. You can validate your JDL using [JDL-Studio](#).
- ⑪ Relationships between entities can be defined in JDL!
- ⑫ If you want pagination on your list screens, you can use infinite scrolling or page links.
- ⑬ You need to specify which microservices contain which entities, or API classes will be generated on your gateway rather than in the microservice.



You can also find this file [on GitHub](#).

Run JHipster's `import-jdl` command to import this microservices architecture definition.

```
jhipster import-jdl apps.jh
```

The project generation process will take a minute or two to run, depending on your internet connection speed and hardware.

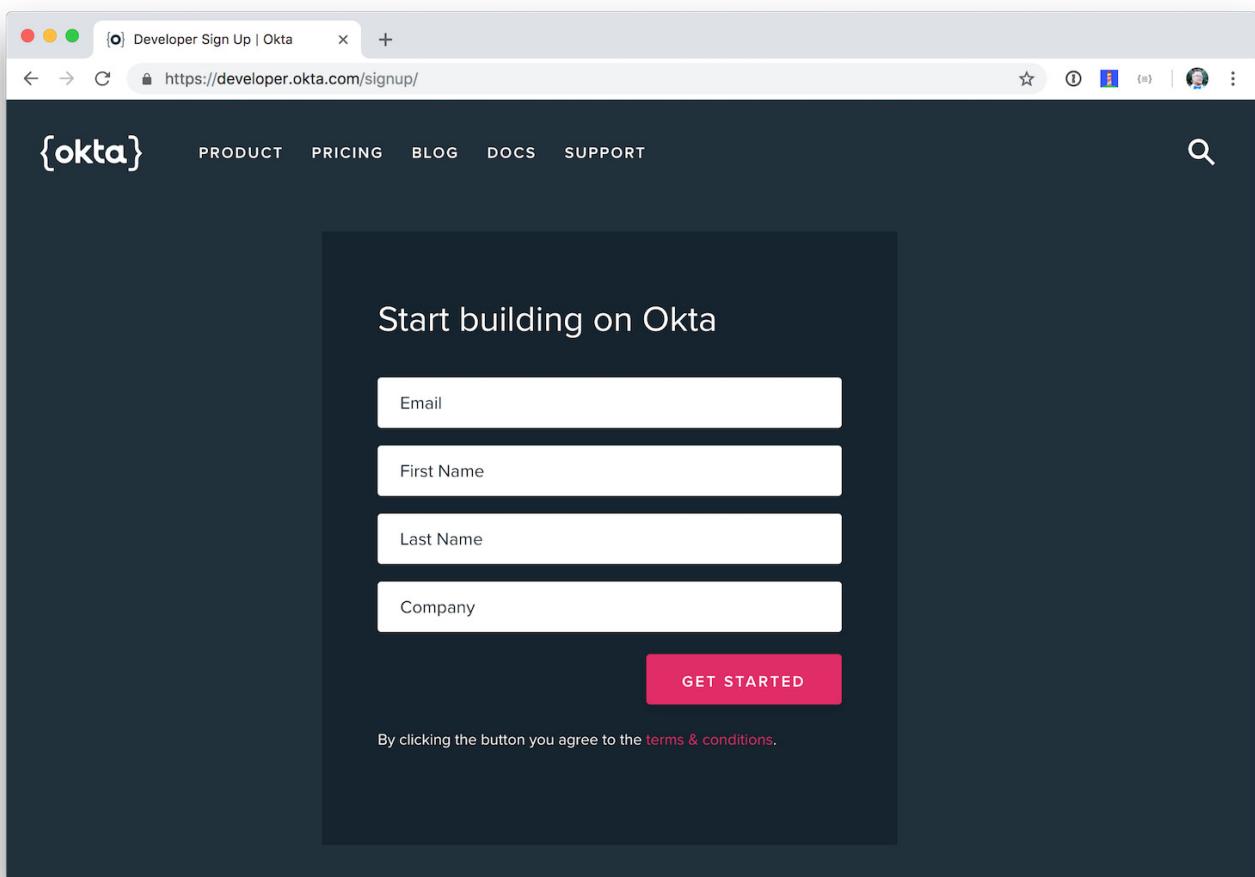
While you're waiting, you can get started with setting up OAuth with Okta.

What is OAuth 2.0?

The OAuth implementation in JHipster leverages Spring Boot and its OAuth 2.0 support (an `@EnableOAuthSso` annotation). OAuth provides single sign-on (SSO) to JHipster applications. “[Securing Microservices with Spring Security OAuth](#)” shows a bare-bones Spring microservices architecture using OAuth. JHipster uses the same internal setup.

JHipster ships with [Keycloak](#) configured for OAuth by default. This works great for local development. However, if you want to deploy your apps to production, you might want to use an identity provider that’s always on, like Okta. Okta offers accounts that are forever-free and allow you 1,000 monthly users at no cost.

To configure your apps to work with Okta, you’ll first need to [create a free developer account](#). After doing so, you’ll get your own Okta domain, which has a name like <https://dev-123456.oktapreview.com>.



Create an OpenID Connect application on Okta

Create an OpenID Connect (OIDC) app in Okta to get a client ID and a secret. This basically means you’re “registering” your application with Okta. Log in to your Okta account and navigate to **Applications > Add Application**. Click **Web** and click **Next**. Give the app a name you’ll remember (e.g.,

JHipster Microservices), and specify `http://localhost:8080/login` as a login redirect URI. Click **Done** and make note of your client ID and client secret values.

In order for the roles coming from Okta to match the default roles in JHipster, you'll need to create them. Create a **ROLE_ADMIN** and **ROLE_USER** group (**Users > Groups > Add Group**) and add users to them. You can use the account you signed up with, or create a new user (**Users > Add Person**). Navigate to **API > Authorization Servers**, click the **Authorization Servers** tab and edit the default one. Click the **Claims** tab and **Add Claim**. Name it `roles`, and include it in the ID Token. Set the value type to **Groups** and set the filter to be a Regex of `.*`.

Modify `gateway/src/main/resources/config/application.yml` to have the following values:

```
security:
  oauth2:
    client:
      access-token-uri: https://{{yourOktaDomain}}/oauth2/default/v1/token
      user-authorization-uri: https://{{yourOktaDomain}}/oauth2/default/v1/authorize
      client-id: {{clientId}}
      client-secret: {{clientSecret}}
      scope: openid profile email
    resource:
      user-info-uri: https://{{yourOktaDomain}}/oauth2/default/v1/userinfo
```

You can also use environment variables to override the default values. I recommend using this technique because 1) you don't need to modify the values in each microservice application and 2) it prevents you from leaking your client secrets in a source-code repository. Create `~/.okta.env` and copy the `export` commands below into it. With that file in place, you can run `source ~/.okta.env` to override the default Spring Security settings.

```
export SECURITY_OAUTH2_CLIENT_ACCESS_TOKEN_URI="https://{{yourOktaDomain}}/oauth2/default/v1/token"
export SECURITY_OAUTH2_CLIENT_USER_AUTHORIZATION_URI="https://{{yourOktaDomain}}/oauth2/default/v1/authorize"
export SECURITY_OAUTH2_RESOURCE_USER_INFO_URI="https://{{yourOktaDomain}}/oauth2/default/v1/userinfo"
export SECURITY_OAUTH2_CLIENT_CLIENT_ID="{{clientId}}"
export SECURITY_OAUTH2_CLIENT_CLIENT_SECRET="{{clientSecret}}"
```



If you want to make Okta settings the default, you can add `source ~/.okta.env` to `~/.bashrc` (or `~/.zshrc`).

If you're hard-coding your Okta settings in `application.yml`, make sure you update your settings in the blog and store apps too. If you're using environment variables, you don't need to make any changes.

If you're using Protractor and want to run your tests against Okta, you'll need to add a user to the `ROLE_ADMIN` group on Okta and change the credentials to match that user in `src/test/javascript/e2e/account/account.spec.ts` and `src/test/javascript/e2e/admin/administration.spec.ts`.

Start JHipster Registry, Keycloak, and MongoDB

You'll need a service-discovery server installed before you can start the gateway. You'll also need Keycloak for authentication and MongoDB for your store microservices. The blog application depends on Elasticsearch and PostgreSQL, but only when running in production mode. Luckily, JHipster creates Docker Compose files for all of the services your apps depend on. You can run the following commands from your project's root directory to start Docker containers for JHipster Registry, Keycloak, and MongoDB.

```
docker-compose -f gateway/src/main/docker/jhipster-registry.yml up -d
docker-compose -f gateway/src/main/docker/keycloak.yml up -d
docker-compose -f store/src/main/docker/mongodb.yml up -d
```

By default, the JHipster Registry will use Keycloak when running with Docker Compose. For everything to work properly, you'll need to add an entry to your `hosts` file for Keycloak.

127.0.0.1	keycloak
-----------	----------

If you want to change it to use Okta, you'll need to modify `gateway/src/main/docker/jhipster-registry.yml` and change the default Keycloak settings to use your Okta settings, or environment variables (recommended).

- `SECURITY_OAUTH2_CLIENT_ACCESS_TOKEN_URI=${SECURITY_OAUTH2_CLIENT_ACCESS_TOKEN_URI}`
- `SECURITY_OAUTH2_CLIENT_USER_AUTHORIZATION_URI=${SECURITY_OAUTH2_CLIENT_USER_AUTHORIZATION_URI}`
- `SECURITY_OAUTH2_CLIENT_CLIENT_ID=${SECURITY_OAUTH2_CLIENT_CLIENT_ID}`
- `SECURITY_OAUTH2_CLIENT_CLIENT_SECRET=${SECURITY_OAUTH2_CLIENT_CLIENT_SECRET}`
- `SECURITY_OAUTH2_RESOURCE_USER_INFO_URI=${SECURITY_OAUTH2_RESOURCE_USER_INFO_URI}`



You can also put these variables in a file and specify an `env_file` setting. See [Environment variables in Compose](#) to learn more.

Then you'll need to restart your JHipster Registry (if it's already running):

```
docker-compose -f gateway/src/main/docker/jhipster-registry.yml restart
```

To log in, you'll need to add <http://localhost:8761/login> as a login-redirect URI in your Okta app.

Run your microservices architecture

Open three terminal windows and navigate to each app ([gateway](#), [blog](#), and [store](#)). In each window, and run Maven to start each app:

```
./mvnw
```



If you already have Maven installed, you can simply use `mvn`.

Open your browser and go to <http://localhost:8761>. Log in, and you should see a welcome page that shows that the gateway and both apps have been registered.

System Status		Instances Registered	
App	Instance ID	Status	disabled
BLOG	blog:5001a90a1431832c07248e5ab87e5704	UP	
GATEWAY	gateway:aef28e9d3e6fc57d71c5bf66a13c3b96	UP	
STORE	store:58d655294403a62d0d44b4dcfda09f26	UP	

General Info		Health	
Total Available Memory	953mb	DiskSpace	UP
Current Memory Usage	127mb (13%)	RefreshScope	UP
Number of CPU	5	DiscoveryComposite	UP
Instance Ip Address	172.19.0.2	ConfigServer	UP
Instance Status	UP	Hystrix	UP

JHipster Registry - www.jhipster.tech

Figure 39. JHipster Registry with gateway registered

Once everything is running, open a browser, go to <http://localhost:8080>, and click **sign in**. You should

be redirected to your Okta tenant to sign in, then back to the gateway once you've entered valid credentials.

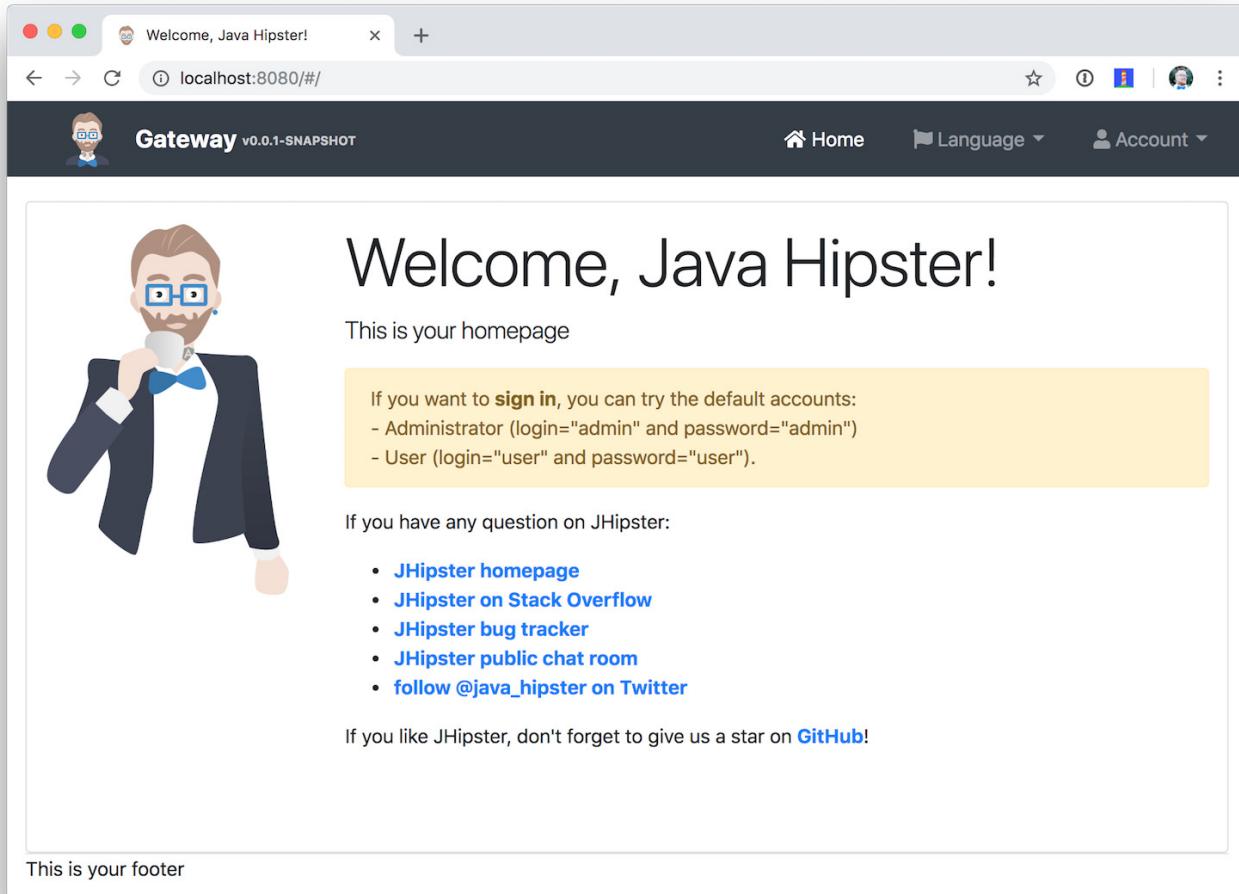


Figure 40. Welcome, JHipster

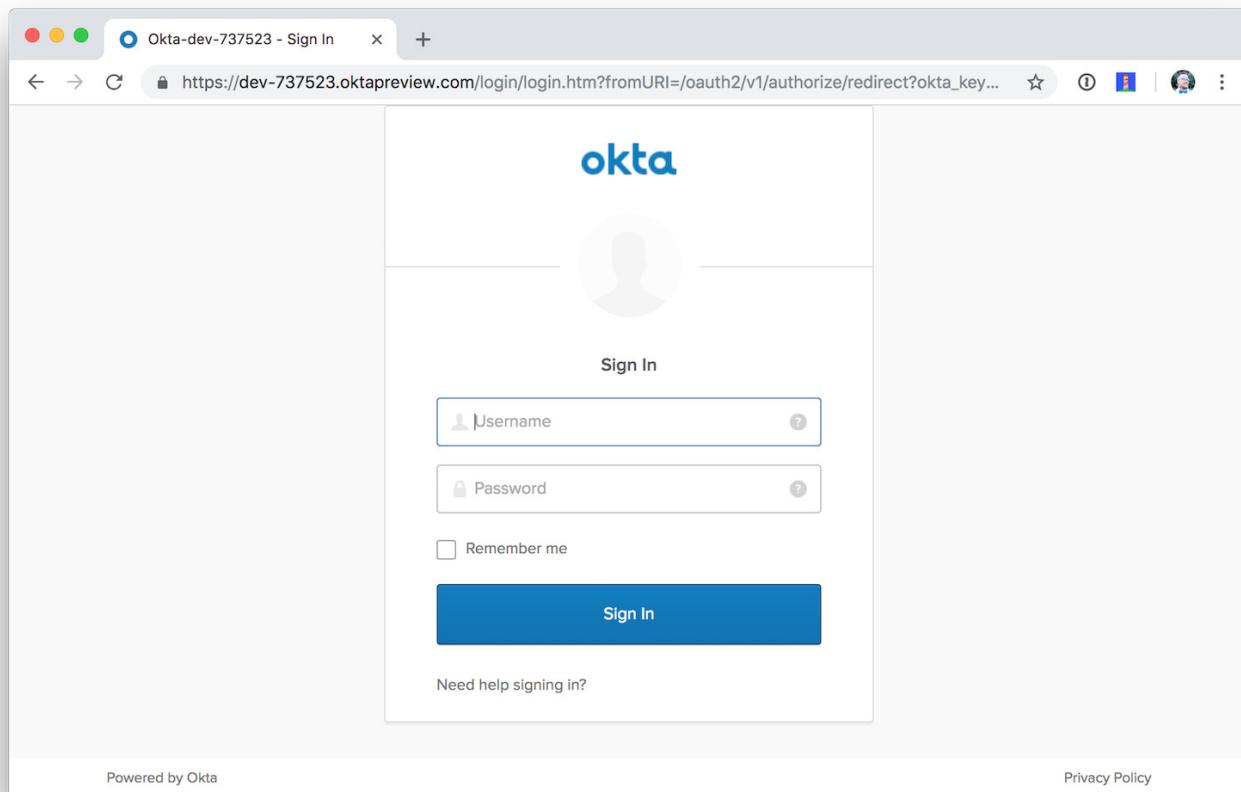


Figure 41. Okta sign-in

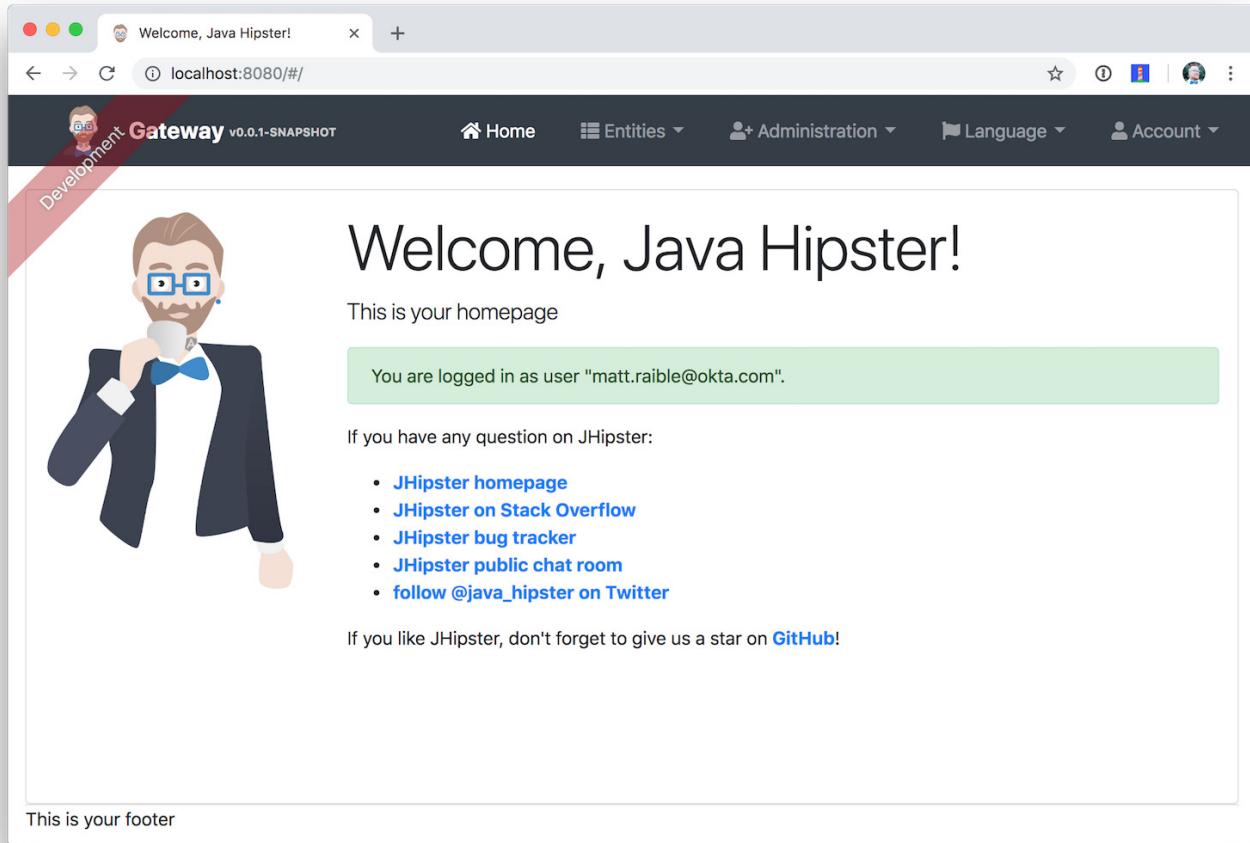


Figure 42. JHipster after Okta SSO

You should be able to navigate to **Entities > Blog** and add a new blog record to your blog microservice.

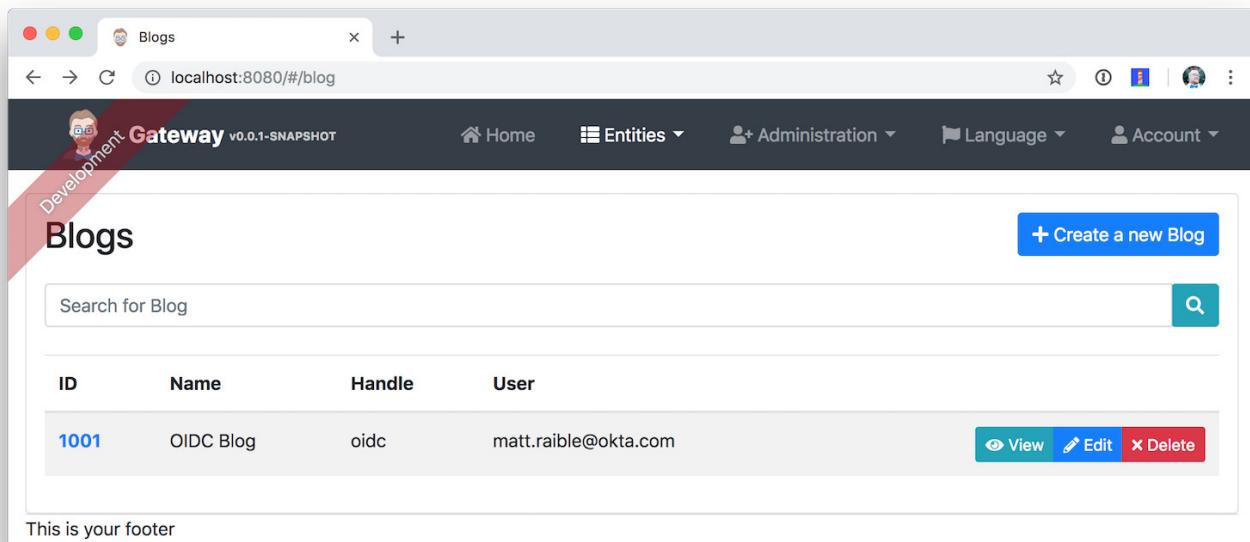


Figure 43. New blog

Navigate to **Entities > Product** to prove your product microservice is working. Since you added an image as a property, you'll be prompted to upload one when creating a new record.

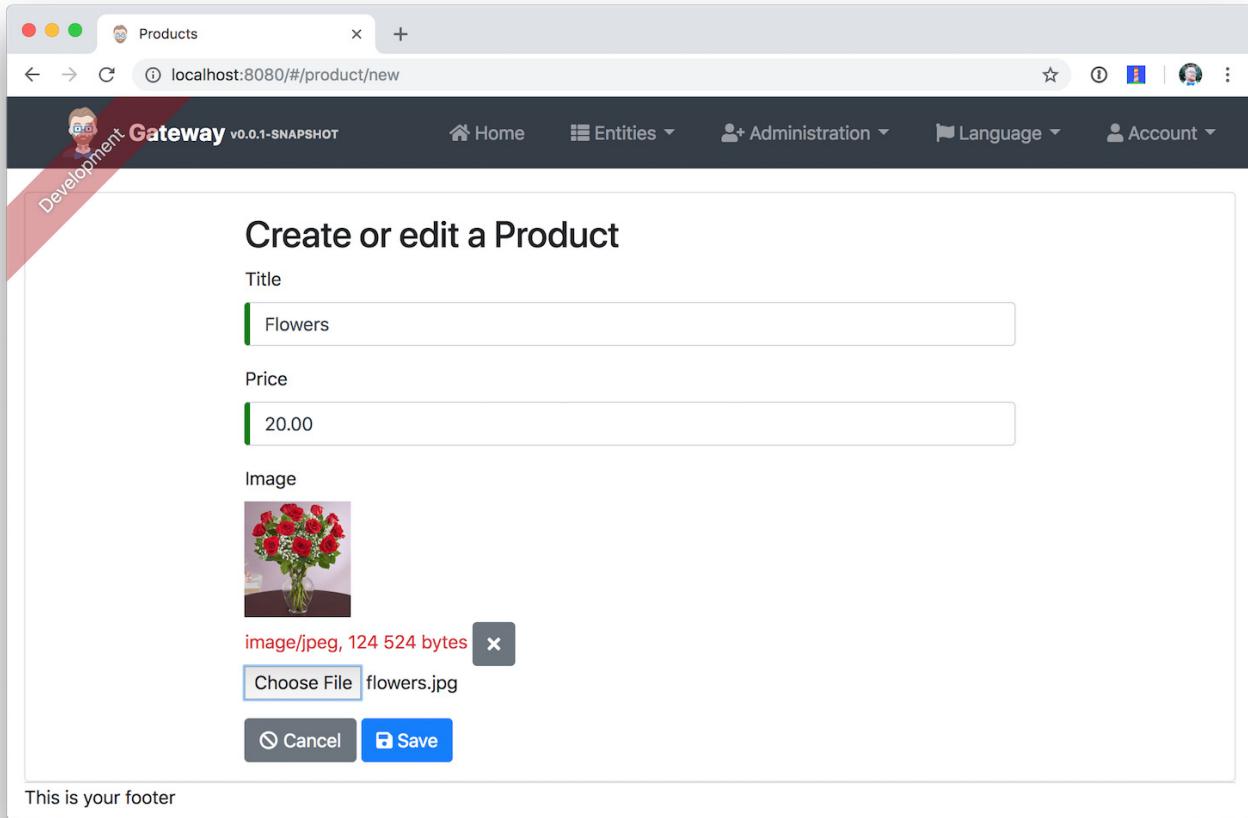


Figure 44. Add product page

Click **Save** and you'll know, based on the generated ID, that it's correctly using MongoDB.

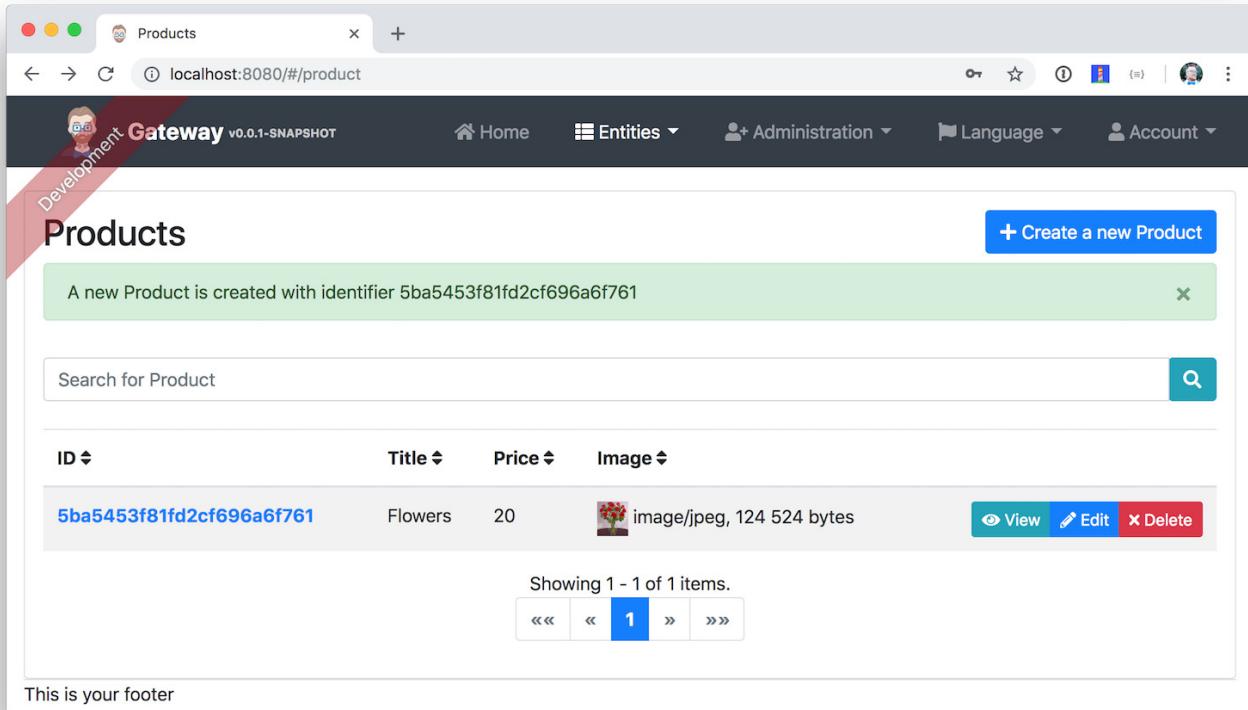


Figure 45. New Product

Use Docker Compose to run everything

Rather than individually starting all your services, you can start them all at once using [Docker Compose](#). To learn more about Docker Compose, see “[A Developer’s Guide to Docker Compose](#)”.

Create a `docker-compose` directory in the root directory ([jhipster-microservices-example](#)) and run JHipster’s Docker Compose subgenerator.

```
mkdir docker-compose
cd docker-compose
jhipster docker-compose
```

Answer as follows when prompted:

Question	Answer
Type of application?	Microservice application

Question	Answer
Type of gateway?	JHipster gateway
Directory location?	.../
Applications to include?	<select all>
Applications with clustered databases?	<blank>
Setup monitoring?	Yes, with JHipster Console
Additional technologies?	Zipkin
Admin password	<choose your own>

You'll get a warning that says you need to generate Docker images by running the following command in the `blog`, `gateway`, and `store` directories. Stop all your running processes and build your Docker images before proceeding.

```
./mvnw package -Pprod jib:dockerBuild
```

While you're waiting for things to build, edit `docker-compose/docker-compose.yml` and change the Spring Security settings from being hard-coded to being environment variables. Make this change for all applications and make sure to add the client ID and secret since those aren't included by default.

```
services:
  blog-app:
    image: blog
    environment:
      - SECURITY_OAUTH2_CLIENT_CLIENT_ID=${SECURITY_OAUTH2_CLIENT_CLIENT_ID}
      - SECURITY_OAUTH2_CLIENT_CLIENT_SECRET=${SECURITY_OAUTH2_CLIENT_CLIENT_SECRET}
      - SECURITY_OAUTH2_RESOURCE_USER_INFO_URI=${SECURITY_OAUTH2_RESOURCE_USER_INFO_URI}
    ...
  gateway-app:
    image: gateway
    environment:
      - SECURITY_OAUTH2_CLIENT_ACCESS_TOKEN_URI=${SECURITY_OAUTH2_CLIENT_ACCESS_TOKEN_URI}
      - SECURITY_OAUTH2_CLIENT_USER_AUTHORIZATION_URI=${SECURITY_OAUTH2_CLIENT_USER_AUTHORIZATION_URI}
      - SECURITY_OAUTH2_CLIENT_CLIENT_ID=${SECURITY_OAUTH2_CLIENT_CLIENT_ID}
      - SECURITY_OAUTH2_CLIENT_CLIENT_SECRET=${SECURITY_OAUTH2_CLIENT_CLIENT_SECRET}
      - SECURITY_OAUTH2_CLIENT_SCOPE=openid profile email
      - SECURITY_OAUTH2_RESOURCE_USER_INFO_URI=${SECURITY_OAUTH2_RESOURCE_USER_INFO_URI}
    ...
  store-app:
    image: store
    environment:
      - SECURITY_OAUTH2_CLIENT_CLIENT_ID=${SECURITY_OAUTH2_CLIENT_CLIENT_ID}
      - SECURITY_OAUTH2_CLIENT_CLIENT_SECRET=${SECURITY_OAUTH2_CLIENT_CLIENT_SECRET}
      - SECURITY_OAUTH2_RESOURCE_USER_INFO_URI=${SECURITY_OAUTH2_RESOURCE_USER_INFO_URI}
```

You can remove Keycloak from `docker-compose/docker-compose.yml` since it won't be used with this configuration.

```
keycloak:
  extends:
    file: keycloak.yml
    service: keycloak
```

You'll need to edit `docker-compose/jhipster-registry.yml` as well.

```

services:
  jhipster-registry:
    ...
    environment:
      - SECURITY_OAUTH2_CLIENT_ACCESS_TOKEN_URI=${SECURITY_OAUTH2_CLIENT_ACCESS_TOKEN_URI}
      - SECURITY_OAUTH2_CLIENT_USER_AUTHORIZATION_URI=${SECURITY_OAUTH2_CLIENT_USER_AUTHORIZATION_URI}
      - SECURITY_OAUTH2_CLIENT_CLIENT_ID=${SECURITY_OAUTH2_CLIENT_CLIENT_ID}
      - SECURITY_OAUTH2_CLIENT_CLIENT_SECRET=${SECURITY_OAUTH2_CLIENT_CLIENT_SECRET}
      - SECURITY_OAUTH2_RESOURCE_USER_INFO_URI=${SECURITY_OAUTH2_RESOURCE_USER_INFO_URI}

```



You can run `docker-compose config` to verify that the environment variables are substituted properly.

When everything has finished building, run `docker-compose up -d` from the `docker-compose` directory. It can take a while to start all 14 containers, so now might be a good time to take a break or get some exercise. You can use Docker's Kitematic to watch the status of your images as they start.



Before you start everything, make sure you've provided adequate CPUs and memory to Docker. It defaults to one CPU and 2 GB of memory — not quite enough for 14 containers!

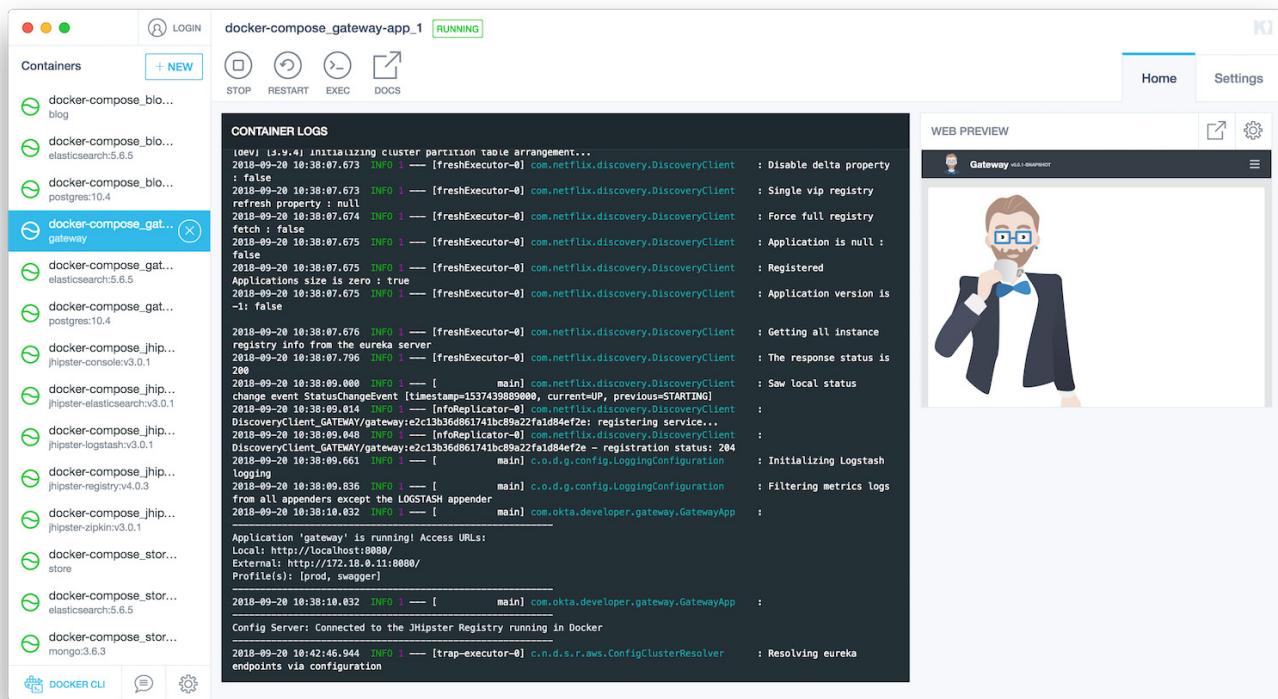


Figure 46. New Product

After you've verified everything works, you can stop all your Docker containers using the following command:

```
docker stop $(docker ps -a -q)
```

If you'd like to remove the images too, you can run:

```
docker rm $(docker ps -a -q)
```

Deploy to Heroku

The founder of JHipster, [Julien Dubois](#), wrote a blog post on the Heroku blog titled “[Bootstrapping Your Microservices Architecture with JHipster and Spring](#)”. Here's an abbreviated set of steps to deploy all your apps to Heroku.

Deploy the JHipster Registry

Heroku and JHipster have configured a JHipster Registry for you, so you just need to click on the button below to start your own JHipster Registry:

 Deploy to Heroku

Enter an app name (I used `okta-jhipster-registry`), add a `JHIPSTER_PASSWORD`, and click **Deploy app**.

Deploy your gateway and apps to Heroku

In each project, run `jhipster heroku` and answer the questions as follows:

Question	Answer
Name to deploy as?	<code><unique-prefix> -<app-name></code> (e.g., okta-gateway, okta-blog, etc.)
Which region?	<code>us</code>

Question	Answer
Type of deployment?	Git
Name of Registry app?	<unique-prefix>-jhipster-registry
JHipster Registry username	admin
JHipster Registry password	<JHIPSTER_PASSWORD from Registry>

When prompted to overwrite files, type **a**.

After each app has finished deploying, you'll want to run the following so they use Okta for authentication.



Make sure you run `source ~/.okta.env` before running the command below. If you don't have the environment variables set beforehand, you'll see an error that says `did not find property 'jhipster.security.client-authorization.client-id'`.

```
heroku config:set \
  SECURITY_OAUTH2_CLIENT_ACCESS_TOKEN_URI="$SECURITY_OAUTH2_CLIENT_ACCESS_TOKEN_URI" \
  SECURITY_OAUTH2_CLIENT_USER_AUTHORIZATION_URI="$SECURITY_OAUTH2_CLIENT_USER_AUTHORIZATION_URI" \
  SECURITY_OAUTH2_RESOURCE_USER_INFO_URI="$SECURITY_OAUTH2_RESOURCE_USER_INFO_URI" \
  SECURITY_OAUTH2_CLIENT_CLIENT_ID="$SECURITY_OAUTH2_CLIENT_CLIENT_ID" \
  SECURITY_OAUTH2_CLIENT_CLIENT_SECRET="$SECURITY_OAUTH2_CLIENT_CLIENT_SECRET"
```

Update your Okta app to have a **Login redirect URI** that matches your Heroku app (e.g., <https://okta-gateway.herokuapp.com/login>). To do this, log in to your Okta account, go to **Applications > JHipster Microservices > General > Edit**.

To see if your apps have started correctly, you can run `heroku logs --tail` in each app's directory. You may see a timeout error, but your app should succeed in starting on your next attempt.

If it crashes and doesn't start, try running `heroku restart`. If that doesn't solve the problem, go to <https://help.heroku.com> and click **Create a ticket** at the top. Click **Running Applications > Java**, scroll to the bottom, and click **Create a ticket**. Enter something like the following for the subject and description, select one of your apps, then submit it.

Subject: JHipster App Startup Timeout

Description: Hello, I have a JHipster (Spring Boot) app that has the following error on startup:

Error R10 (Boot timeout) -> Web process failed to bind to \$PORT within 90 seconds of launch

The URL is:

* <https://<your-app>-store.herokuapp.com/>

Can you please increase the timeout on this app?

Thanks!

Below are screenshots to prove everything worked after I deployed to Heroku. ☺

The screenshot shows the JHipster Registry v4.0.4 dashboard running on Heroku. The top navigation bar includes links for Home, Eureka, Configuration, Administration, and Help. The main content area is divided into four sections:

- System Status:** Displays environment information (JHipster-Environment), data center (JHipster-DataCenter), current time (2018-10-12T03:53:12 +0000), system uptime (00:56), and below renew threshold (false).
- Instances Registered:** Lists registered applications: BLOG, GATEWAY, and STORE, each with its instance ID and a green "UP" status indicator.
- General Info:** Shows total available memory (230mb), current memory usage (111mb (48%)), number of CPU (8), instance IP address (172.16.45.2), and instance status (UP).
- Health:** Monitors various services: DiskSpace, RefreshScope, DiscoveryComposite, ConfigServer, and Hystrix, all of which are marked as UP.

Figure 47. JHipster Registry on Heroku

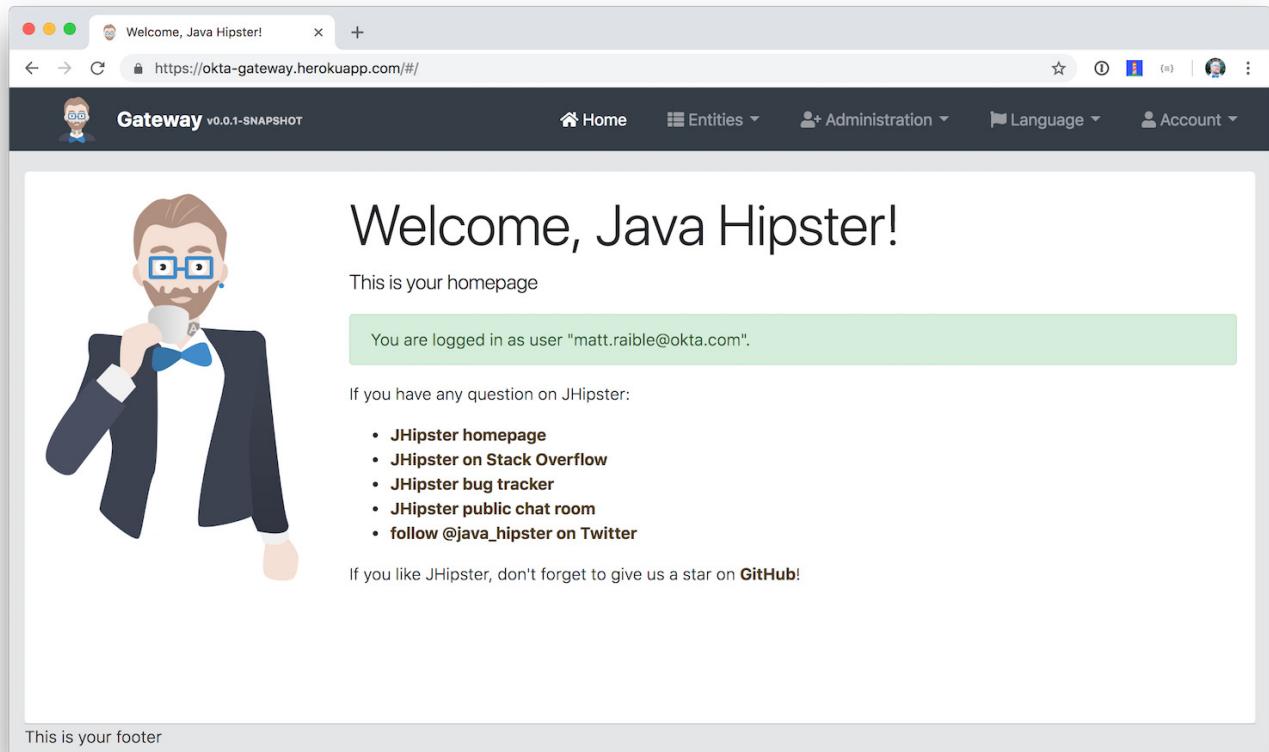


Figure 48. Successful login on Heroku

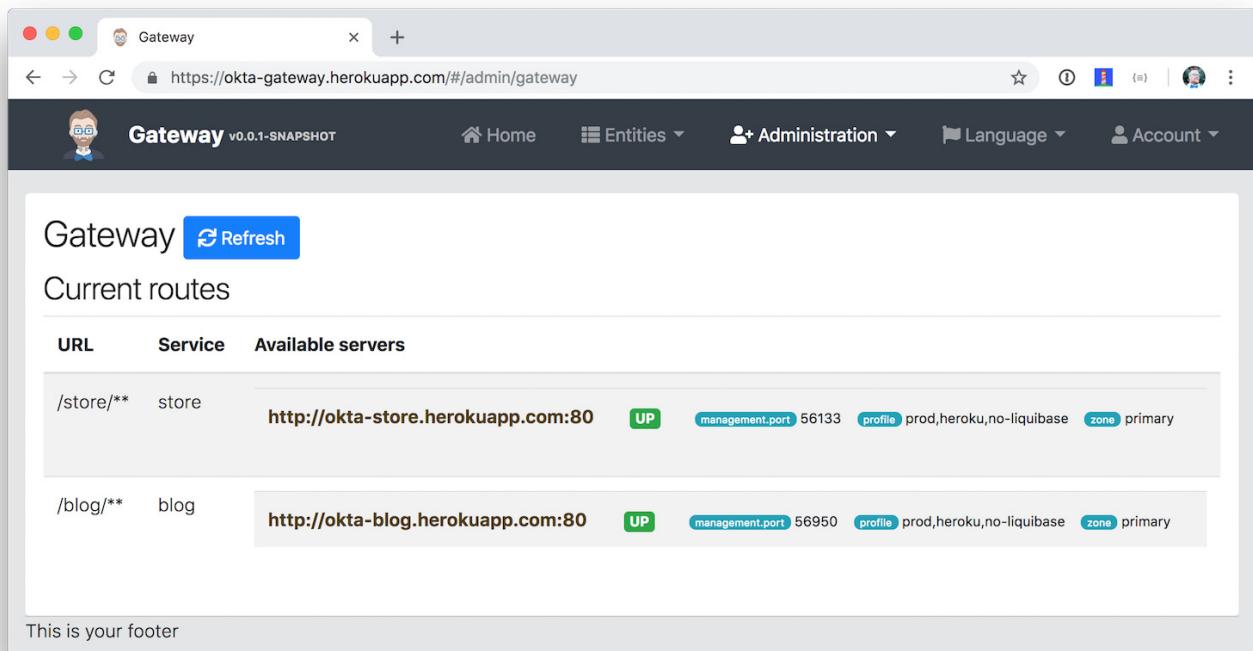


Figure 49. Heroku gateway routes

ID	Name	Handle	User
1001	JHipster on Heroku	jhipster	matt.raible@okta.com

+ Create a new Blog

Search for Blog

This is your footer

Figure 50. Blog on Heroku

ID	Title	Price	Image
5ba47937b6c2150004436152	Flowers on Heroku	42	image/jpeg, 124 524 bytes

+ Create a new Product

Search for Product

Showing 1 - 1 of 1 items.

This is your footer

Figure 51. Store on Heroku



If you're interested in deploying to Google Cloud using Kubernetes, you might enjoy my blog post and screencast on how to “[Build JHipster Microservices and Deploy to Google Cloud with Kubernetes](#)”.

Source code

You can find the source code for this microservices example at <https://github.com/oktadeveloper/okta-jhipster-microservices-oauth-example>.

Summary

I hope you've enjoyed this whirlwind tour of how to create a microservices architecture with JHipster. Just because JHipster makes microservices easy doesn't mean you should use them. Using a microservices architecture is a great way to scale development teams, but if you don't have a large team, a "Majestic Monolith" might work better.

If you'd like to learn more about microservices, authentication, and JHipster, see the following resources.

- ["Build a Microservices Architecture for Microbrews with Spring Boot"](#)
- ["Secure a Spring Microservices Architecture with Spring Security and OAuth"](#)
- ["Develop and Deploy Microservices with JHipster"](#) (uses JWT for authentication)
- ["Use OpenID Connect Support with JHipster"](#)
- ["Hello Istio 🛡, welcome to JHipster"](#)

Action!

I hope you've enjoyed learning how JHipster can help you develop hip web applications! It's a nifty project, with an easy-to-use entity generator, a pretty UI and many Spring Boot best-practice patterns. The project team follows five simple [policies](#), paraphrased here:

1. The development team votes on policies.
2. JHipster uses technologies with their default configurations as much as possible.
3. Only add options when there is sufficient added value in the generated code.
4. For the Java code, follow the default IntelliJ IDEA formatting and coding guidelines.
5. Use strict versions for third-party libraries.

These policies help the project maintain its sharp edge and streamline its development process. If you have features you'd like to add or if you'd like to refine existing features, please follow the project and help with its development and support. We're always looking for help!

Now that you've learned how to use Angular, Bootstrap, and Spring Boot with JHipster, go forth and develop awesome applications!

Additional reading

If you want to learn more, here are some suggestions.

Deepu K Sasidharan and Sendil Kumar N, two prolific committers on the JHipster team, wrote [*Full Stack Development with JHipster*](#) (Packt Publishing, March 2018).

The definitive book about Spring's JDBC, JPA, and NoSQL support is [*Spring Data: Modern Data Access for Enterprise Java*](#) by Mark Pollack *et al.* (O'Reilly Media, October 2012).

Learn how to use Spring Boot to build apps faster than ever before with [*Learning Spring Boot 2.0*](#) by Greg L. Turnquist (Packt Publishing, November 2017).

An in-depth and up-to-date book on Angular is [*ng-book: The Complete Book on Angular*](#) by Nathan Murray, Felipe Coury, Ari Lerner, and Carlos Taborda (Fullstack.io, November 2018).

Learn how to develop cloud-ready JVM applications and microservices across a variety of environments with [*Cloud Native Java: Designing Resilient Systems with Spring Boot, Spring Cloud, and Cloud Foundry*](#) by Josh Long and Kenny Bastani (O'Reilly Media, August 2017).

About the author

Matt Raible is a Java Hipster. He grew up in the backwoods of Montana with no electricity or running water. He walked a mile and a half to the bus stop on school days. His mom and sister often led the early-morning hikes, but his BMX skills overcame this handicap later in life.

He started writing HTML, CSS, and JavaScript in the early '90s and got into Java in the late '90s. He loves the Volkswagen Bus, like no one should love anything. He has a passion for skiing, mountain biking, VWs, and good beer. Matt is married to an awesome woman and amazing photographer, Trish McGinity. They love skiing, rafting, and camping with their fun-loving kids, Abbie and Jack.



Matt's blog is at raibledesigns.com. You can also find him on Twitter @mraible. Matt drives a 1966 21-Window Bus and a 1990 Vanagon Syncro.

