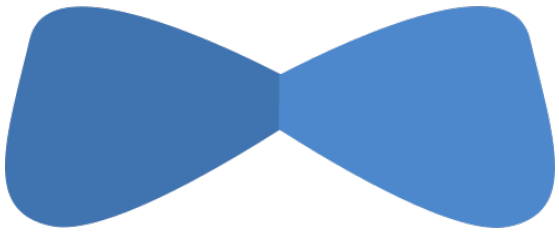
THE JHIPSTER MINI-BOOK

Matt Raible

InfoQ

The JHipster Mini-Book



Matt Raible

Version 7.0.2 | 2023-05-30

The JHipster Mini-Book

Published by C4Media, publisher of InfoQ.com.

Production Editor: Ana Ciobotaru
Copy Editor: Lawrence Nyveen and Maureen Spencer
Cover and Interior Design: Dragos Balasoiu

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

Table of Contents

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!

Acknowledgments

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 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 editors, Lawrence Nyveen and Maureen Spencer, were a tremendous help in correcting my words and making this book easier to read. Thanks, Laurie and Maureen!

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. In early 2013, I first used AngularJS on a project where I implemented 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 an HTML5 UI in a native iOS application. We used AngularJS, Bootstrap, and Spring Boot in that project, which worked very well for us.

When I heard about JHipster, I was motivated to use it immediately. 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 well. I wanted to further my knowledge of this wonderful project and show Java developers that they could be hip again by leveraging Angular and Bootstrap. 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. Version 5 for JHipster v5 was released in November 2018. This version is updated for JHipster version 7. 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 and 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, you should have a fundamental understanding of the technology, including when and where to apply it, how it relates to other technologies, and an assimilated knowledge of other professionals who know what this technology is about. You will then be able to make intelligent decisions about the technology once your projects require it and can delve into more detailed sources (such as larger books or tutorials) at that time.

Who is this book 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.

What you need for this book

To try code samples, you will need a computer running an up-to-date operating system (Windows, Linux, or macOS). You will need Node.js and Java installed. The book's code was tested using Node.js 16 and Java 11.

Conventions

We use several 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.

Listing 1. src/app/search/search.component.html

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

Listing 2. src/main/java/demo/DemoApplication.java

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

    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 constructor'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. Tell us 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. According to the 2021 Stack Overflow Developer Survey, Spring and Angular were among the top 10 web frameworks used by professional developers. In 2022, Angular ranked as the fifth most used web framework, while Spring moved to an *other frameworks* category and ranked fourth.

Julien Dubois started JHipster in October 2013 (Julien’s first commit was on October 21, 2013). The first public release (version 0.3.1) launched on December 7, 2013. Since then, the project has had over 240 releases! It is an open-source, Apache 2.0-licensed project on GitHub. It has a core team of 38 developers and over 660 contributors. You can find its homepage at www.jhipster.tech. Its GitHub project shows it’s mostly written in JavaScript (53%), TypeScript (19%), and Java (17%). EJS is trailing in the fourth position with 8%.

JHipster started as 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 provides everything needed to begin 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.)
- the package manager (npm/Yarn)

JHipster now uses its own `jhipster` CLI to replace `yo`. This makes for a better developer experience because you can simply type `jhipster` instead of `yo jhipster`.

This book shows you how to build an app with JHipster and guides you through many tools, techniques, and options. Furthermore, it explains the UI components and API building blocks so that 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. I recently finished the project and 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, 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 wanted to develop for a while.

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

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

My goal is to earn 15 points per week. If I earn more, I'll likely lose weight and have good blood pressure. If I earn 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.

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

always important. Wearables that can track your health stats can 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 many 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 11 installed, as well as Maven and Git. I installed Node.js 16 from Nodejs.org, then ran the following command to install JHipster.

```
npm install -g generator-jhipster@7
```

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 JHipster prevents this with a validation error: `Your base name cannot contain special characters or a blank space`.

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.

Listing 3. yo-rc.json

```
{
  "generator-jhipster": {
    "applicationType": "monolith",
    "authenticationType": "jwt",
    "baseName": "TwentyOnePoints",
    "blueprints": [],
    "buildTool": "gradle",
    "cacheProvider": "ehcache",
    "clientFramework": "angularX",
```

```
    "clientPackageManager": "npm",
    "clientTheme": "none",
    "clientThemeVariant": "",
    "creationTimestamp": 1662997089611,
    "cypressAudit": false,
    "cypressCoverage": false,
    "databaseType": "sql",
    "devDatabaseType": "h2Disk",
    "devServerPort": 4200,
    "dtoSuffix": "DTO",
    "enableGradleEnterprise": false,
    "enableHibernateCache": true,
    "enableSwaggerCodegen": false,
    "enableTranslation": true,
    "entitySuffix": "",
    "jhiPrefix": "jhi",
    "jhipsterVersion": "7.9.3",
    "languages": ["en", "fr"],
    "messageBroker": false,
    "microfrontend": false,
    "microfrontends": [],
    "nativeLanguage": "en",
    "otherModules": [],
    "packageName": "org.jhipster.health",
    "pages": [],
    "prodDatabaseType": "postgresql",
    "reactive": false,
    "searchEngine": "elasticsearch",
    "serverPort": "8080",
    "serverSideOptions": ["searchEngine:elasticsearch"],
    "serviceDiscoveryType": "no",
    "skipCheckLengthOfIdentifier": false,
    "skipClient": false,
    "skipFakeData": false,
    "skipUserManagement": false,
    "testFrameworks": ["cypress"],
    "websocket": false,
    "withAdminUi": true
  }
}
```


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 can undo changes to your database, either automatically or with custom SQL.

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

```
BUILD SUCCESSFUL in 1m 12s
15 actionable tasks: 13 executed, 2 up-to-date
```

JHipster 5+ 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
```

Next, I started the app using `./gradlew` and then ran the UI integration tests in another terminal with `npm run e2e`. All tests passed with flying colors.

```
$ npm run e2e

> twenty-one-points@0.0.1-SNAPSHOT e2e
> npm run e2e:cypress:headed --

> twenty-one-points@0.0.1-SNAPSHOT e2e:cypress:headed
> npm run e2e:cypress -- --headed

> twenty-one-points@0.0.1-SNAPSHOT e2e:cypress
> cypress run --e2e --browser chrome --record ${CYPRESS_ENABLE_RECORD:-false}

...

    -  All specs passed!

Execution time: 41 s.
```

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://127.0.0.1:8080/
        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. I shut down the Docker image running PostgreSQL.

```
docker-compose -f src/main/docker/postgresql.yml down
```

Then, I tried creating a local PostgreSQL database with settings from `application-prod.yml`.

```
psql (14.5)
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=# \q
```

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 things were not set up correctly.

```
$ ./gradlew -Pprod
...

org.postgresql.util.PSQLException: FATAL: role "TwentyOnePoints" does not exist
```

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 GitHub.

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 GitHub repository, then pushed everything. I created the repository on GitHub before executing these commands.

```
git remote add origin git@github.com:mraible/21-points.git
git branch -M main
git push -u origin main
```

The response should indicate success:

```
Enumerating objects: 652, done.
Counting objects: 100% (652/652), done.
Delta compression using up to 10 threads
Compressing objects: 100% (618/618), done.
Writing objects: 100% (652/652), 952.77 KiB | 11.34 MiB/s, done.
Total 652 (delta 79), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (79/79), done.
To github.com:mraible/21-points.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

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, 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 several 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.

Listing 4. src/main/webapp/content/scss/_bootstrap-variables.scss

```
/*
 * Bootstrap overrides https://getbootstrap.com/docs/5.1/customize/sass/
 * All values defined in bootstrap source
 * https://github.com/twbs/bootstrap/blob/v5.1.3/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 from
https://mdbootstrap.com/docs/standard/
// https://github.com/mdbootstrap/mdb-ui-
kit/blob/master/src/scss/bootstrap/_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;
$headings-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: 3;
  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 Material Design for Bootstrap and Angular with JHipster

If you'd like to use Material Design for Bootstrap & Angular with JHipster, that's possible too.

1. Install The MDB Angular UI KIT:

   ```
   npm i mdb-angular-ui-kit@3 @angular/cdk@14
   ```

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 the MDB stylesheet in `src/main/webapp/content/scss/vendor.scss`:

   ```
   @import url('https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&display=swap');
   @import '~mdb-angular-ui-kit/assets/scss/mdb.scss';
   ```

5. Add `https://fonts.googleapis.com` and `https://fonts.gstatic.com` to the `style-src` and `font-src` content security policy rules in `src/main/resources/config/application.yml`.

   ```
   jhipster:
     ...
     security:
       content-security-policy: "... style-src 'self' 'unsafe-inline'
   https://fonts.googleapis.com; ... font-src 'self' data:
   https://fonts.gstatic.com"
   ```

6. 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;
}
```

7. Modify the `.dropdown-menu` rule to set the display to `none`.

```
.dropdown-menu {
  padding-left: 0px;
  display: none;
}
```

Below is a screenshot taken after these changes.

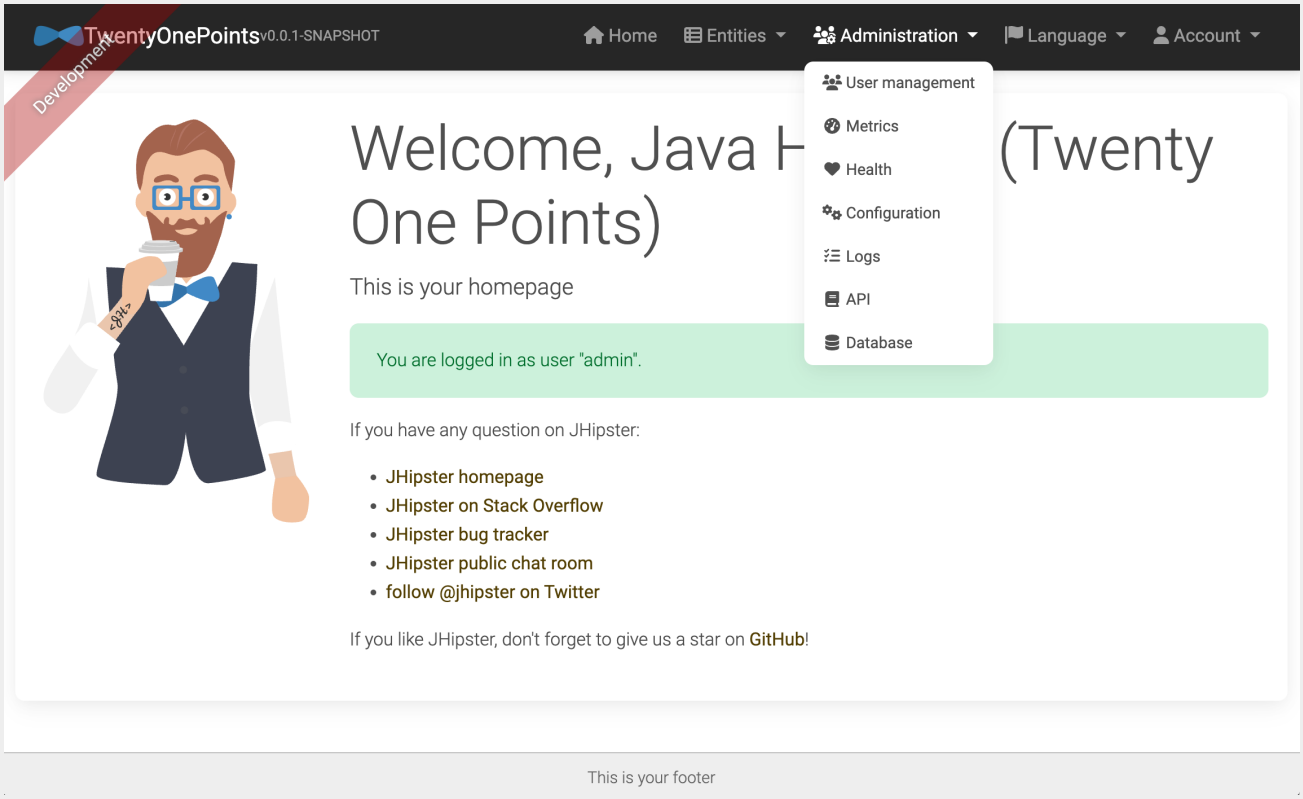


Figure 2. JHipster with Bootstrap Angular Material

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
- a HTML page

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

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 and 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, and GenMyModel. 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 7 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, eaten 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. 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.

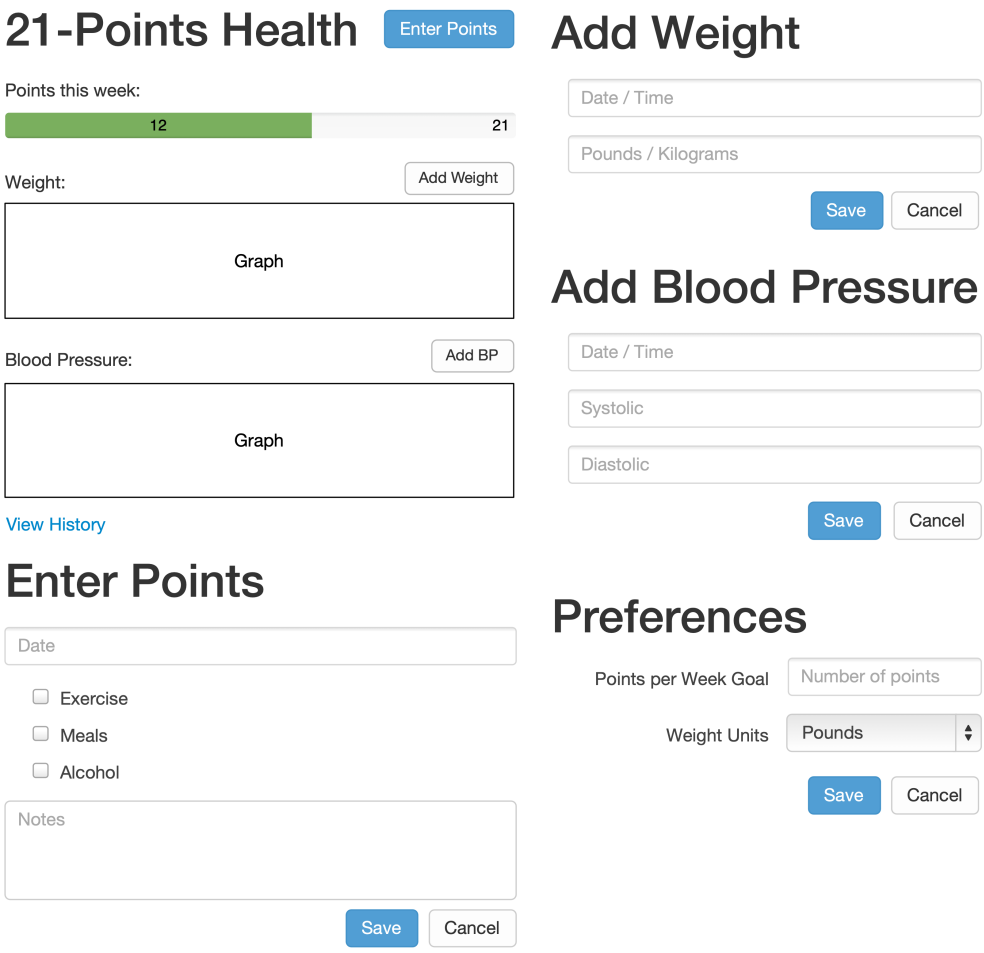


Figure 3. UI mockup

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 Q1 2023). 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.

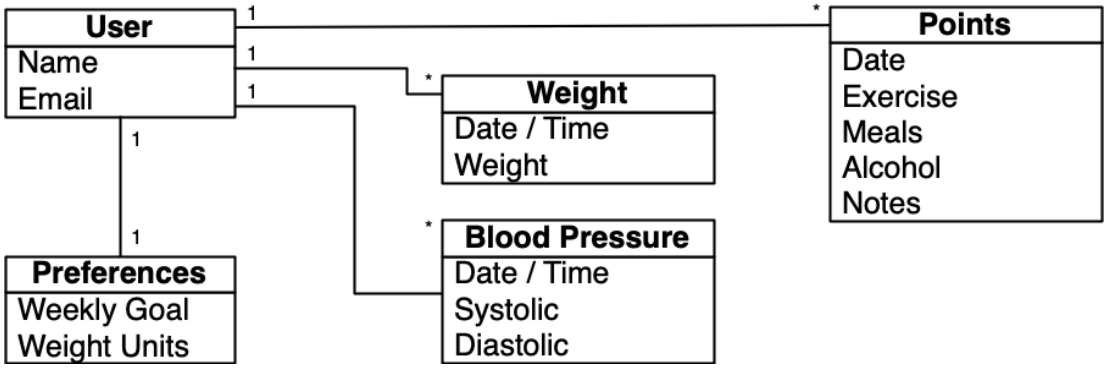


Figure 4. 21-Points Health entity diagram

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
? Is this entity read-only? No
? Do you want pagination and sorting on your entity? Yes, with pagination links and sorting headers

     info Creating changelog for entities Points
    ...
    force .yo-rc-global.json
    force .yo-rc.json
    force .jhipster/Points.json
   create src/test/java/org/jhipster/health/domain/PointsTest.java
   create src/test/java/org/jhipster/health/web/rest/PointsResourceIT.java
   create src/main/java/org/jhipster/health/web/rest/PointsResource.java
   create src/main/webapp/app/entities/points/points.model.ts
   create src/main/java/org/jhipster/health/repository/PointsRepository.java
   create src/main/java/org/jhipster/health/repository/search/PointsSearchRepository.java
   create src/main/webapp/app/entities/points/points.module.ts
   create src/main/webapp/app/entities/points/list/points.component.html
   create src/main/webapp/app/entities/points/points.test-samples.ts
   create src/main/webapp/app/entities/points/detail/points-detail.component.html
   create src/main/webapp/app/entities/points/list/points.component.ts
   create src/main/webapp/app/entities/points/route/points-routing.module.ts
   create src/main/webapp/app/entities/points/detail/points-detail.component.ts
   create src/main/webapp/app/entities/points/route/points-routing-resolve.service.ts
   create src/main/webapp/app/entities/points/service/points.service.ts
   create src/main/webapp/app/entities/points/update/points-form.service.ts
   create src/main/webapp/app/entities/points/update/points-form.service.spec.ts
   create src/main/webapp/app/entities/points/update/points-update.component.html
   create src/main/webapp/app/entities/points/update/points-update.component.ts
   create src/main/webapp/app/entities/points/delete/points-delete-dialog.component.html
   create src/main/webapp/app/entities/points/detail/points-detail.component.spec.ts
   create src/main/webapp/app/entities/points/delete/points-delete-dialog.component.ts
   create src/main/webapp/app/entities/points/list/points.component.spec.ts
   create src/main/webapp/app/entities/points/route/points-routing-resolve.service.spec.ts
   create src/main/webapp/app/entities/points/service/points.service.spec.ts
   create src/main/webapp/app/entities/points/update/points-update.component.spec.ts
   create src/main/webapp/app/entities/points/delete/points-delete-dialog.component.spec.ts
   create src/main/webapp/i18n/en/points.json
   create src/main/webapp/i18n/fr/points.json
   create src/test/javascript/cypress/e2e/entity/points.cy.ts
 conflict src/main/webapp/i18n/en/global.json
? Overwrite src/main/webapp/i18n/en/global.json? overwrite this and all others
    force src/main/webapp/i18n/en/global.json
   create src/main/resources/config/liquibase/changelog/20221108000520_added_entity_Points.xml
   create src/main/resources/config/liquibase/changelog/20221108000520_added_entity_constraints_Points.xml
   create src/main/resources/config/liquibase/fake-data/points.csv
```

```
   create src/main/java/org/jhipster/health/domain/Points.java
    force src/main/webapp/i18n/fr/global.json
    force src/main/java/org/jhipster/health/config/CacheConfiguration.java
    force src/main/webapp/app/entities/entity-routing.module.ts
    force src/main/resources/config/liquibase/master.xml
    force src/main/webapp/app/layouts/navbar/navbar.component.html
    force .yo-rc.json
    force .jhipster/Points.json

No change to package.json was detected. No package manager install will be executed.
Entity Points generated successfully.

Running `webapp:build` to update client app
```

You can see the validation rules for the date and notes above, but you don't see how I created the relationship with the 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
? 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 1m 25s
```

I checked in seven changed files and 144 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 several 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 forms
- improved titles and button labels by editing generated JSON files in `src/main/webapp/i18n/en`
- adjusted date format in custom DatePipe's to use `MMM D, YYYY` instead of `D MMM YYYY`
- defaulted to the current date on new entries
- replaced point metrics with icons on list/detail screens
- replaced point metrics with checkboxes on the update screen

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

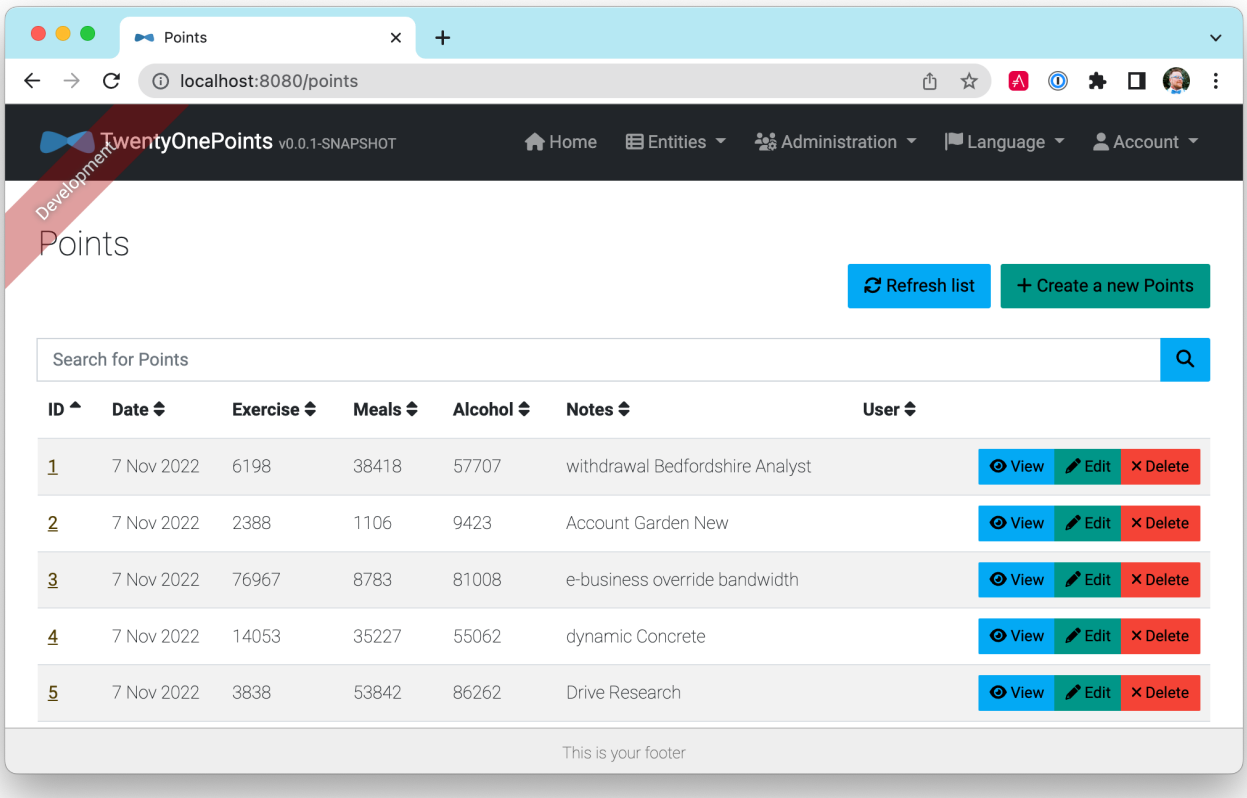


Figure 5. Default Daily Points list

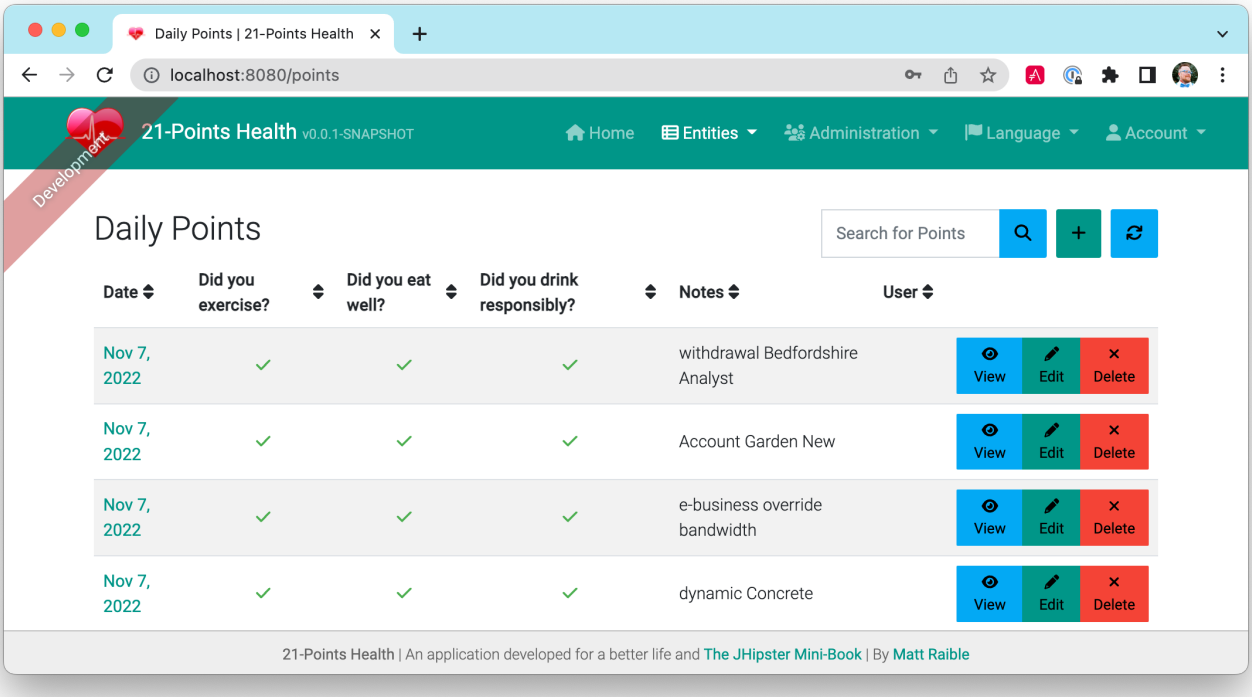


Figure 6. Default Daily Points list after UI improvements

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.

Listing 5. src/main/webapp/app/entities/points/list/points.component.html

```
<div class="row">
  <jhi-alert-error></jhi-alert-error>
  <jhi-alert></jhi-alert>

  <div class="col-md-8 col-sm-4">
    <h2 id="page-heading" data-cy="PointsHeading">
      <span jhiTranslate="twentyOnePointsApp.points.home.title">Points</span>
    </h2>
  </div>
  <div class="col-md-4 col-sm-8 text-right d-flex flex-row-reverse">
    <button class="btn btn-info ms-2" (click)="load()" [disabled]="isLoading"
            [ngbTooltip]="refreshTooltip" placement="bottom">
      <fa-icon icon="sync" [spin]="isLoading"></fa-icon>
      <ng-template #refreshTooltip>
        <span jhiTranslate="twentyOnePointsApp.points.home.refreshListLabel">Refresh list</span>
      </ng-template>
    </button>

    <button
      id="jh-create-entity"
      data-cy="entityCreateButton"
      class="btn btn-primary jh-create-entity create-points ms-2"
      [routerLink]="['/points/new']"
      [ngbTooltip]="addTooltip"
      placement="bottom"
    >
      <fa-icon icon="plus"></fa-icon>
      <ng-template #addTooltip>
        <span class="hidden-sm-down"
              jhiTranslate="twentyOnePointsApp.points.home.createLabel">Add Points</span>
      </ng-template>
    </button>
    <form name="searchForm" class="w-100">
      <div class="input-group h-100">
        <label class="visually-hidden" for="currentSearch"
               jhiTranslate="twentyOnePointsApp.points.home.search">Search for Points</label>
        <input
          type="text"
          class="form-control"
          [(ngModel)]="currentSearch"
          id="currentSearch"
          name="currentSearch"
          placeholder="{{ 'twentyOnePointsApp.points.home.search' | translate }}"
```

```
        />

        <button class="btn btn-info" (click)="search(currentSearch)">
          <fa-icon icon="search"></fa-icon>
        </button>

        <button class="btn btn-danger" (click)="search('')" *ngIf="currentSearch">
          <fa-icon icon="trash-alt"></fa-icon>
        </button>
      </div>
    </form>
  </div>
</div>
<div class="row">
  ...
</div>
```

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

Listing 6. src/main/webapp/app/entities/points/list/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>
```

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

Listing 7. src/main/webapp/app/entities/points/update/points-update.component.html

```
<div class="form-check">
  <label class="form-check-label" jhiTranslate="twentyOnePointsApp.points.exercise"
         for="field_exercise">Exercise</label>
  <input type="checkbox" class="form-check-input" name="exercise" id="field_exercise"
         data-cy="exercise" formControlName="exercise"/>
</div>
<div class="form-check">
  <label class="form-check-label" jhiTranslate="twentyOnePointsApp.points.meals"
         for="field_meals">Meals</label>
```

```
  <input type="checkbox" class="form-check-input" name="meals" id="field_meals"
         data-cy="meals" formControlName="meals" />
</div>
<div class="form-check mb-3">
  <label class="form-check-label" jhiTranslate="twentyOnePointsApp.points.alcohol"
         for="field_alcohol">Alcohol</label>
  <input type="checkbox" class="form-check-input" name="alcohol" id="field_alcohol"
         data-cy="alcohol" formControlName="alcohol" />
</div>
```

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

Listing 8. src/main/webapp/app/entities/points/update/points-update.component.ts

```
save(): void {
  this.isSaving = true;
  const points = this.pointsFormService.getPoints(this.editForm);

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

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

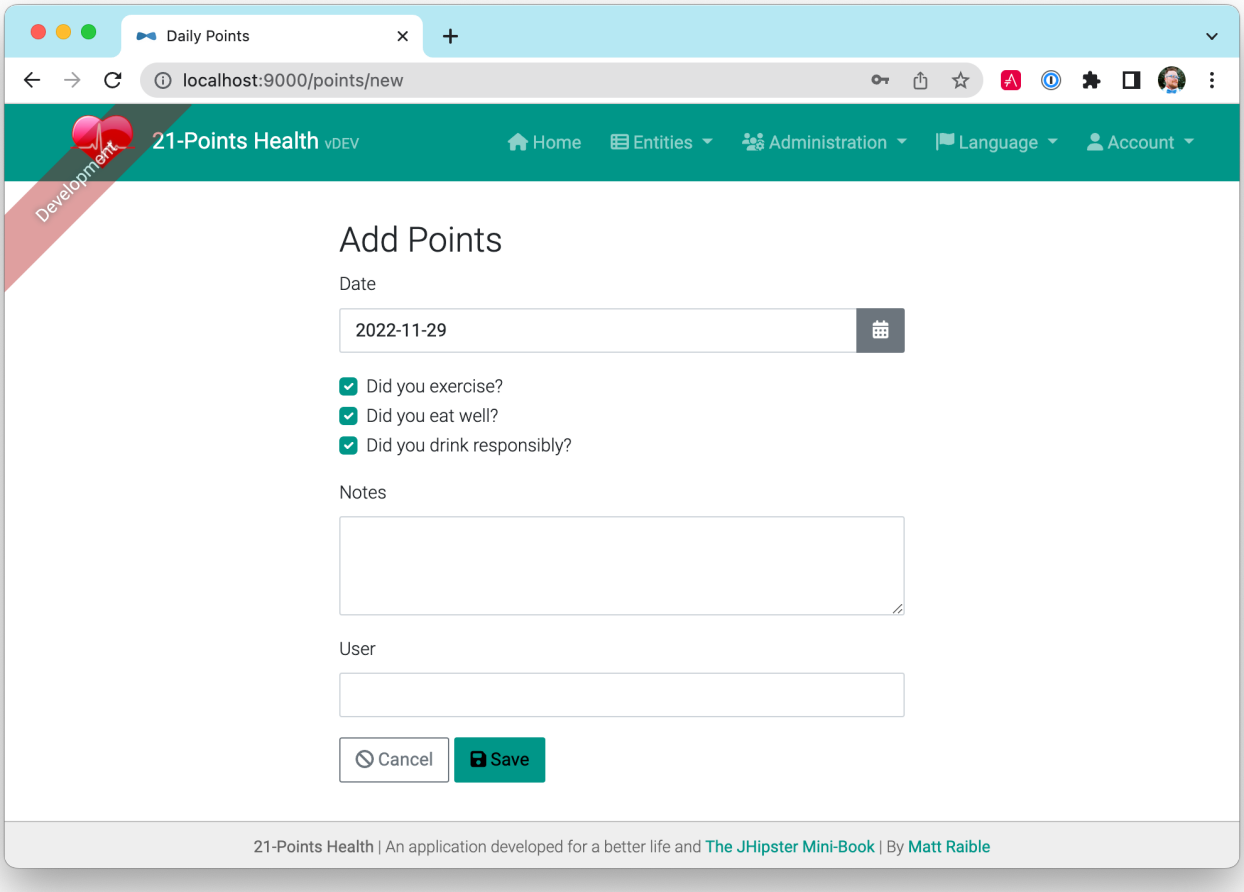


Figure 7. Add Points page

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 data

I wanted to make several improvements to what users could see based on their roles. Users 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 ensure 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.

Listing 9. src/main/webapp/app/entities/points/points-update.component.html

```
<div class="form-group" *jhiHasAnyAuthority="'ROLE_ADMIN'">
  <label class="form-label" jhiTranslate="twentyOnePointsApp.points.user"
         for="field_user">User</label>
  <select class="form-control" id="field_user" data-cy="user" name="user"
          formControlName="user" [compareWith]="compareUser">
    <option [ngValue]="null"></option>
    <option [ngValue]="userOption"
            *ngFor="let userOption of usersSharedCollection">{{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 I needed to make to `PointsResource.java`.

Listing 10. 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;

public class PointsResource {

     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")
     public ResponseEntity<Points> createPoints(@Valid @RequestBody Points points) throws
URISyntaxException {
         log.debug("REST request to save Points : {}", points);
         if (points.getId() != null) {
             throw new BadRequestAlertException("A new points cannot already have an ID",
ENTITY_NAME, "idexists");
```

```
         }
+        if (!SecurityUtils.hasCurrentUserThisAuthority(AuthoritiesConstants.ADMIN)) {
+            log.debug("No user passed in, using current user: {}",
SecurityUtils.getCurrentUserLogin().get());
+            String username = SecurityUtils.getCurrentUserLogin().get();
+            points.setUser(userRepository.findOneByLogin(username).get());
+        }
         Points result = pointsRepository.save(points);
         pointsSearchRepository.index(result);
         return ResponseEntity
             .created(new URI("/api/points/" + result.getId()))
             .headers(HeaderUtil.createEntityCreationAlert(applicationName, true,
ENTITY_NAME, result.getId().toString()))
             .body(result);
     }
}
```

`SecurityUtils` is a class JHipster provides when you create a project. The integration test for this class, `PointsResourceIT.java`, has a `@WithMockUser` annotation on it. This means it's security-aware, and its tests will pass without changes.

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 the current user. Admin users should see all users' data. To facilitate this feature, I modified `PointsResource#getAllPoints()` to have a switch based on the user's role. Rather than showing you the diff of method, here's the whole thing.

Listing 11. src/main/java/org/jhipster/health/web/rest/PointsResource.java

```
@GetMapping("/points")
public ResponseEntity<List<Points>> getAllPoints(
    @org.springdoc.api.annotations.ParameterObject Pageable pageable,
    @RequestParam(required = false, defaultValue = "false") boolean eagerload
) {
    log.debug("REST request to get a page of Points");
    Page<Points> page;
    if (SecurityUtils.hasCurrentUserThisAuthority(AuthoritiesConstants.ADMIN)) {
        page = pointsRepository.findAllByOrderByDateDesc(pageable);
    } else {
        page = pointsRepository.findByUserIsCurrentUser(pageable);
    }
    HttpHeaders headers = PaginationUtil.generatePaginationHttpHeaders(
        ServletUriComponentsBuilder.fromCurrentRequest(), page);
    return ResponseEntity.ok().headers(headers).body(page.getContent());
```

```
}
```

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>`.

Listing 12. 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.

Listing 13. 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 add `@WithMockUser` to `PointsResourceIT#getAllPoints()` to run the test as an administrator.

Listing 14. src/test/java/org/jhipster/health/web/rest/PointsResourceIT.java

```
@Test
@Transactional
@WithMockUser(authorities = AuthoritiesConstants.ADMIN)
void getAllPoints() throws Exception { ... }
```

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 the 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>
```

Because `home.component.html` already contains `<jhi-alert></jhi-alert>`, I didn't have to do anything else to get success messages to show up on the homepage.

Points this week

To get points achieved in the current week, I started by adding a unit test to `PointsResourceIT.java` that would allow me to prove my API was working.

Listing 15. src/test/java/org/jhipster/health/web/rest/PointsResourceIT.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("$", 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.

Listing 16. 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.

Listing 17. 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`.

Listing 18. src/main/java/org/jhipster/health/web/rest/PointsResource.java

```
/**
 * {@code GET  /points-this-week} : get all the points for the current week
 *
 * @param timezone the user's timezone
 * @return the {@link ResponseEntity} with status {@code 200 (OK)}
*          and a count of points in body.
```

```
 */
@GetMapping("/points-this-week")
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().get()
    );
    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 `IPointsPerWeek` interface in `points.model.ts`:

Listing 19. src/main/webapp/app/entities/points/points.model.ts

```
export interface IPointsPerWeek {
  week?: dayjs.Dayjs;
  points: number;
}
```

I imported it into `points.service.ts` and added a few new methods:

Listing 20. src/main/webapp/app/entities/points/service/points.service.ts

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

protected convertWeekResponseFromServer(res: HttpResponse<IPointsPerWeek>):
  HttpResponse<IPointsPerWeek> {
  return res.clone({
    body: res.body ? this.convertWeekDateFromServer(res.body) : null,
  });
}

protected convertWeekDateFromServer(pointsPerWeek: IPointsPerWeek):
  IPointsPerWeek {
  return {
    ...pointsPerWeek,
    week: dayjs(pointsPerWeek.week),
  };
}
```

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

Listing 21. src/main/webapp/app/home/home.component.ts

```
import { PointsService } from '../entities/points/service/points.service';
import { IPointsPerWeek } from '../entities/points/points.model';

...
export class HomeComponent implements OnInit, OnDestroy {
  account: Account | null = null;
  pointsThisWeek: IPointsPerWeek = { points: 0 };
  pointsPercentage?: number;

  private readonly destroy$ = new Subject<void>();

  constructor(private accountService: AccountService, private router: Router,
              private pointsService: PointsService) {
  }

  ngOnInit(): void {
    this.accountService
```

```
      .getAuthenticationState()
      .pipe(takeUntil(this.destroy$))
      .subscribe(account => {
        this.account = account;
        this.getUserData(); // this line is new
      });
  }

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

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

Listing 22. src/main/webapp/app/home/home.component.html

```
<div class="row">
  <div class="col-md-11 col-xs-12 mt-1">
    <ngb-progressbar [max]="21" [value]="pointsThisWeek.points"
                     [hidden]="!pointsThisWeek.points" [striped]="true">
      <span *ngIf="pointsThisWeek.points" class="fw-bolder">
          {{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.

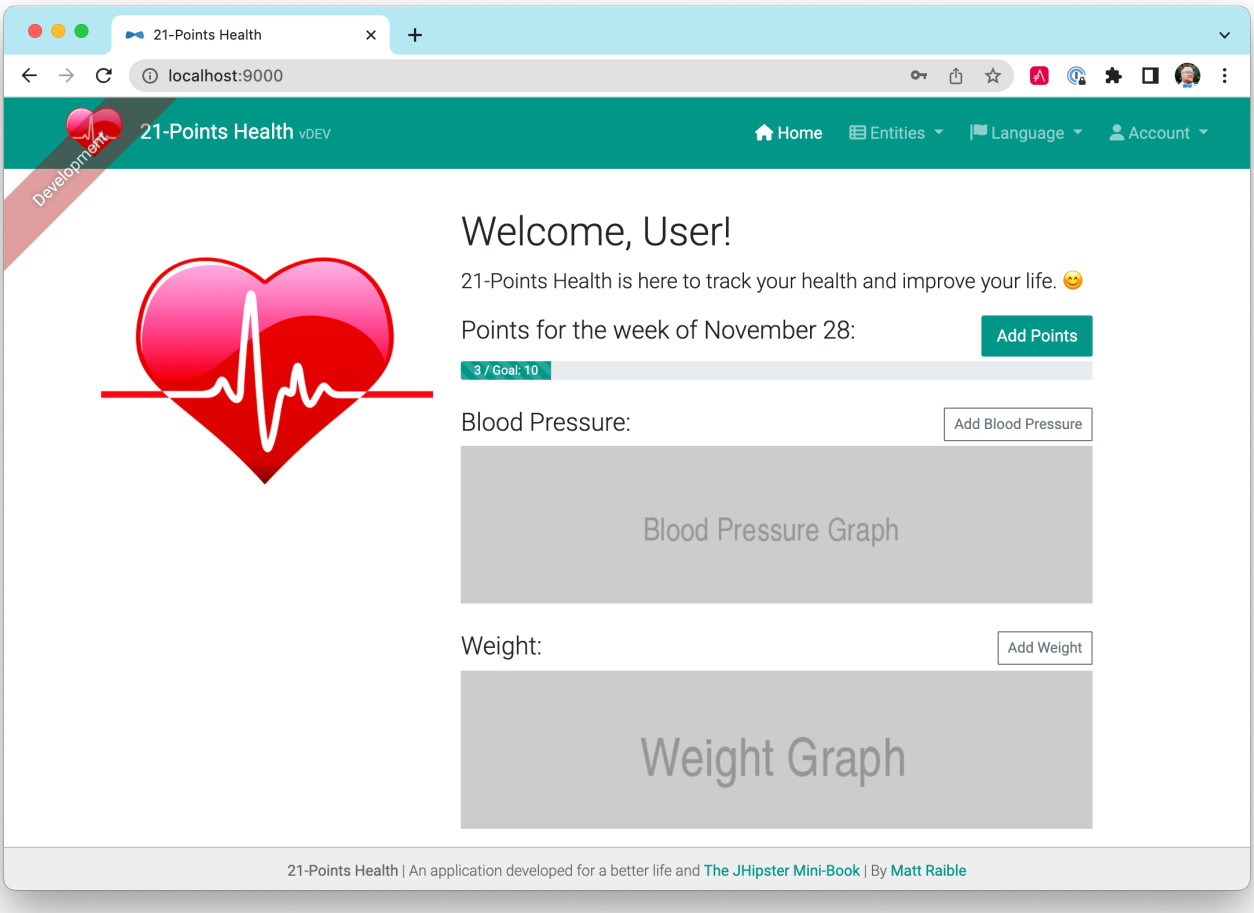


Figure 8. Progress bar for points this week

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 accessing a user's preferences easier, I modified `PreferencesRepository.java` and added a method to retrieve a user's preferences.

Listing 23. 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).

Listing 24. src/main/java/org/jhipster/health/web/rest/PreferencesResource.java

```
/**
```

```
 * {@code GET  /my-preferences} : get the current user's preferences
 *
 * @return the preferences or default (weeklyGoal: 10) if none exist.
 */
@GetMapping("/my-preferences")
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 `preferences.service.ts` in the client.

Listing 25. src/main/webapp/app/entities/preferences/service/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 logic to calculate the background color of the progress bar.

Listing 26. src/main/webapp/app/home/home.component.ts

```
export class HomeComponent implements OnInit, OnDestroy {
  account: Account | null = null;
  pointsThisWeek: IPointsPerWeek = {points: 0};
  pointsPercentage?: number;
  preferences!: IPreferences;

  private readonly destroy$ = new Subject<void>();

  constructor(private accountService: AccountService, private router: Router,
              private pointsService: PointsService,
              private preferencesService: PreferencesService) {
```

```
  }

  ngOnInit(): void { ... }

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

      // Get points for the current week
      this.pointsService.thisWeek().subscribe(response => {
          if (response.body) {
            this.pointsThisWeek = response.body;
            this.pointsPercentage =
              (this.pointsThisWeek.points / 21) * 100;

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

Listing 27. src/main/webapp/app/home/home.component.html

```
<ngb-progressbar [max]="21" [value]="pointsThisWeek.points"
                 [hidden]="!pointsThisWeek.points" [striped]="true">
  <span *ngIf="pointsThisWeek.points" class="fw-bolder">
    {{ pointsThisWeek.points }} / Goal: {{ preferences.weeklyGoal }}
  </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>
```

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

Listing 28. src/main/webapp/app/home/home.component.html

```
<a [routerLink]="['/preferences' +
   (preferences && preferences.id ? '/' + preferences.id + '/edit' : '/new')]"
   class="float-end" 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 `BloodPressureResourceIT.java` to set up my expectations.

Listing 29. src/test/java/org/jhipster/health/web/rest/BloodPressureResourceIT.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);

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

    // Get the blood pressure readings for the last 30 days
    restBloodPressureMockMvc.perform(get("/api/bp-by-days/{days}", 30))
        .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.

Listing 30. 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;
    }

    // getters and setters and toString() generated by IntelliJ
}
```

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 the date entered.

Listing 31. 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.

Listing 32. src/main/java/org/jhipster/health/web/rest/BloodPressureResource.java

```
/**
 * {@code GET  /bp-by-days/:days} : get all the blood pressure readings by last x days.
 *
 * @param days the number of days.
 * @return the {@link ResponseEntity} with status {@code 200 (OK)}
 *         and with body the {@link BloodPressureByPeriod}.
 */
@RequestMapping(value = "/bp-by-days/{days}")
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.getCurrentUserLogin().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`:

Listing 33. src/main/java/org/jhipster/health/repository/BloodPressureRepository.java

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

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

Listing 34. 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.findAllByTimestampBetweenAndUserLoginOrderByTimestampAsc(
            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`.

Listing 35. src/main/webapp/app/entities/blood-pressure/service/blood-pressure.service.ts

```
last30Days(): Observable<HttpResponse<IBloodPressureByPeriod>> {
  return this.http.get<IBloodPressureByPeriod>('api/bp-by-days/30',
                                               { observe: 'response' });
}
```

This required adding a new `IBloodPressureByPeriod` interface in `blood-pressure.model.ts` and importing it in `blood-pressure.service.ts`:

Listing 36. src/main/webapp/app/entities/blood-pressure/blood-pressure.model.ts

```
export interface IBloodPressureByPeriod {
  period: string;
  readings: Array<IBloodPressure>;
}
```

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

In the first three versions of this book, I looked for an Angular library that integrated with D3.js and found ng2-nvd3. However, this library is no longer maintained. I chose Chart.js and ng2-charts for Angular integration.

```
npm install -E ng2-charts@4.0.0
```

Then I updated `home.module.ts` to import the `NgChartsModule`.

Listing 37. src/main/webapp/app/home/home.module.ts

```
import { NgChartsModule } from 'ng2-charts';
```

```
@NgModule({
  imports: [..., NgChartsModule],
  declarations: [HomeComponent],
})
export class HomeModule {}
```

I modified `home.component.ts` to have the `BloodPressureService` as a dependency and went to work building the data so Chart.js could render it.

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

Listing 38. 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;

  if (bpReadings.readings.length) {
    this.bpOptions = {
      plugins: {
        legend: { display: true },
        title: {
          display: true,
          text: bpReadings.period,
        },
      },
      scales: {
        y: {
          beginAtZero: false,
        },
        x: {
          beginAtZero: false,
        },
      },
    };
    const labels: any = [];
    const systolics: any = [];
    const diastolics: any = [];
    const upperValues: any = [];
    const lowerValues: any = [];
    bpReadings.readings.forEach((item: IBloodPressure) => {
      const timestamp = dayjs(item.timestamp).format('MMM DD');
      labels.push(timestamp);
```

```
      systolics.push({
        x: timestamp,
        y: item.systolic,
      });
      diastolics.push({
        x: timestamp,
        y: item.diastolic,
      });
      upperValues.push(item.systolic);
      lowerValues.push(item.diastolic);
    });
    const datasets = [
      {
        data: systolics,
        label: 'Systolic',
      },
      {
        data: diastolics,
        label: 'Diastolic',
      },
    ];
    this.bpData = {
      labels,
      datasets,
    };
    // set y scale to be 10 more than max and min
    this.bpOptions.scales = {
      y: {
        max: Math.max(...upperValues) + 10,
        min: Math.min(...lowerValues) - 10,
      },
    };
    // show both systolic and diastolic on hover
    this.bpOptions.interaction = {
      mode: 'index',
      intersect: false,
    };
  } else {
    this.bpReadings.readings = [];
  }
});
```

Finally, I used a `<canvas>` element with a "baseChart" attribute in `home.component.html` to read `bpOptions` and `bpData`, then display a chart.

Listing 39. src/main/webapp/app/home/home.component.html

```
<div class="row mt-1">
  <div class="col-md-11 col-xs-12">
    <canvas
      baseChart
      *ngIf="bpReadings && bpReadings.readings.length"
      height="125"
      [type]="'line'"
      [data]="bpData"
      [options]="bpOptions"
    >
    </canvas>
    <ngb-alert [dismissible]="false"
               [hidden]="bpReadings && 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.

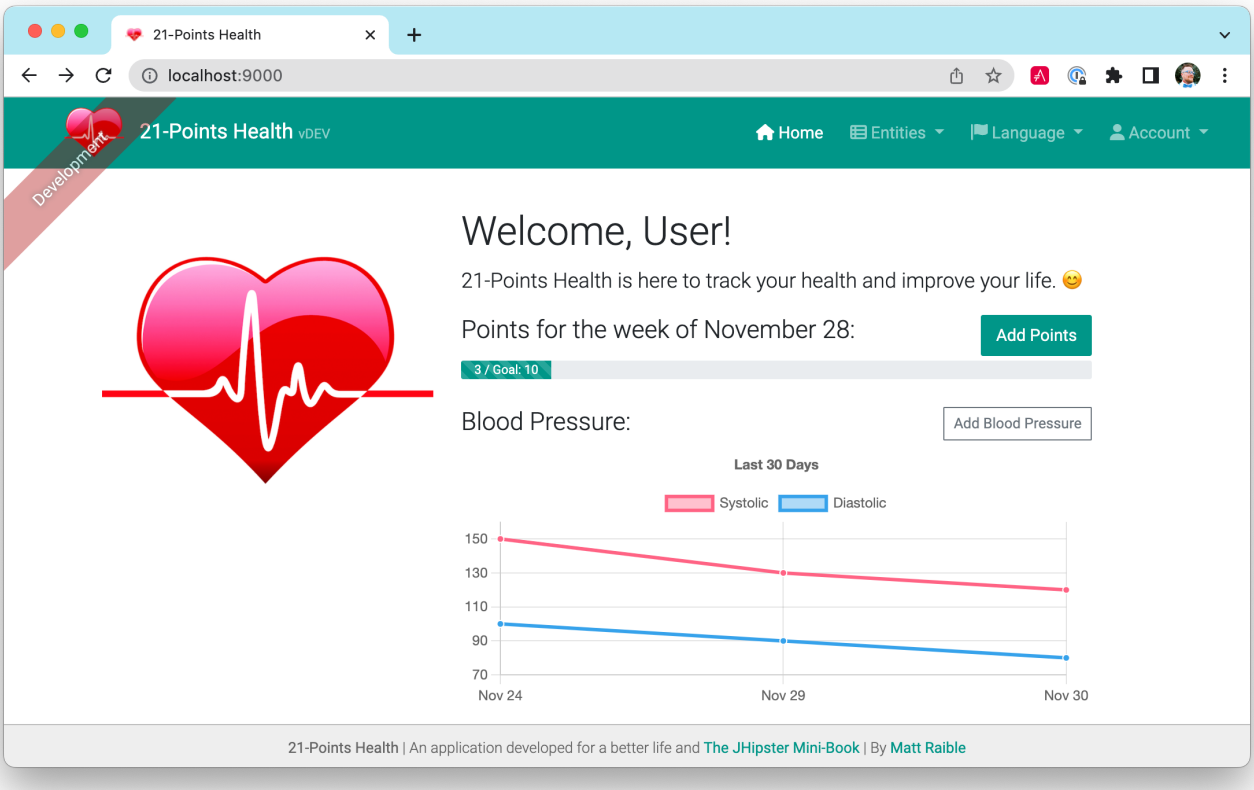


Figure 9. Chart of blood pressure during the last 30 days

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 1,291 lines of code. JHipster did the rest for me, generating 98.3% of the code in my project!

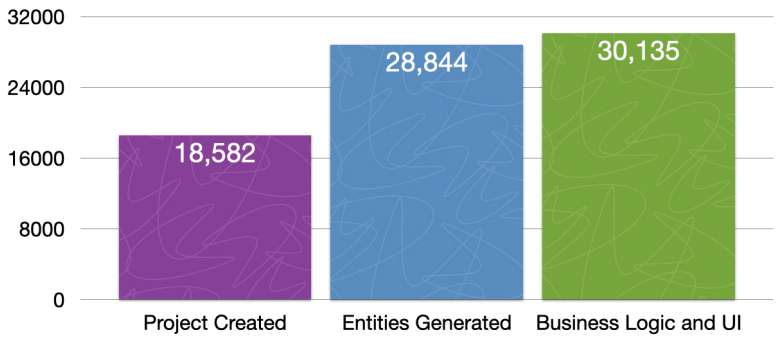


Figure 10. Project lines of code

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

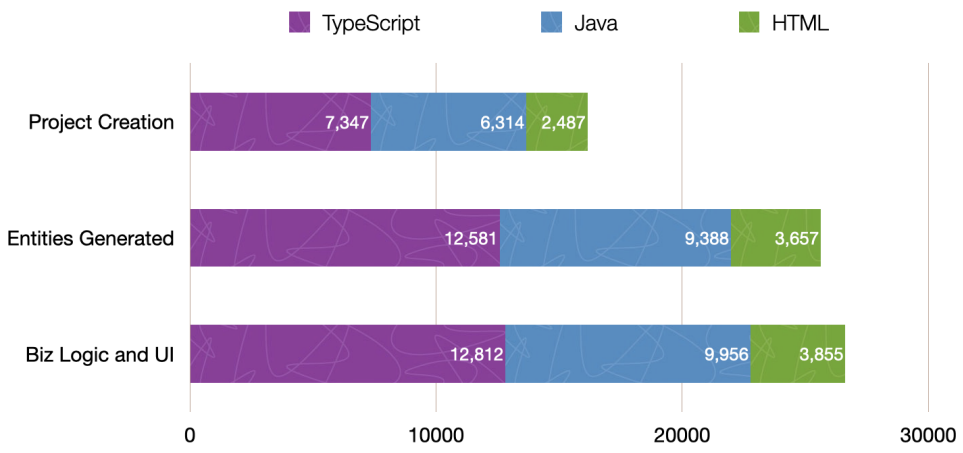


Figure 11. Project lines of code by language

The amount of code I had to write in each language was 231 lines of TypeScript, 568 lines of Java, and 351 lines of HTML. The other 294 lines were a mix of various other files.

Wahoo! Thanks, JHipster!

Testing

You probably noticed that most of the Java code I wrote was for the tests. I felt these tests were essential to prove that the business logic I implemented was correct. It's never easy to work with dates but Java 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 make them work before writing tests. It's usually a 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 Jest, and I like to write integration tests with Cypress.

I did not show any UI tests in this section, but JHipster generated a bunch for me. Running `npm test` shows 87.35% of lines are covered in the UI!

Deploying to Heroku

JHipster ships with support for deploying to Google App Engine, Heroku, and Kubernetes, including Microsoft Azure, AWS, Google Cloud, and Digital Ocean. 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.

```
./gradle bootJar -Pprod
```

The command looks similar when using Maven.

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

The production profile is used to build an optimized JavaScript client. You can invoke this using webpack by running `npm run webapp:build:prod`. The production profile also configures gzip compression with a servlet filter, cache headers, and monitoring via Micrometer. If you have a Prometheus 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 I had a default JHipster application with no entities.

```
$ heroku login
heroku: Press any key to open the browser to login or q to exit:
Opening browser to https://cli-auth.heroku.com/auth/cli/browser/57c43ff8...
Logging in... done
Logged in as matt@raibledesigns.com
```

I ran `jhipster heroku` as recommended in the Deploying to Heroku documentation. When prompted, I tried using the name “21points” for my application.

```
$ 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)
? Which Java version would you like to use to build and run your app ? 11

Using existing Git repository

Installing Heroku CLI deployment plugin

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

You can see my first attempt failed for the same reason that creating the initial JHipster app failed: it

didn't like the app name to start 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.

I ran `git checkout .yo-rc.json` to revert the changes the Heroku sub-generator made, then tried again. I typed "a" when prompted to overwrite `build.gradle`.

```
$ jhipster heroku
...
Heroku configuration is starting
? Name to deploy as: health-by-points
? On which region do you want to deploy ? us
? Which type of deployment do you want ? Git (compile on Heroku)
? Which Java version would you like to use to build and run your app ? 11

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

Provisioning bonsai elasticsearch addon
Provisioning database addon heroku-postgresql --as DATABASE
No suitable cache addon for cacheprovider ehcache available.

Creating Heroku deployment files
    force .yo-rc-global.json
    force .yo-rc.json
   create Procfile
   create system.properties
   create gradle/heroku.gradle
 conflict build.gradle
? Overwrite build.gradle? (ynarxdeiH) a
? Overwrite build.gradle? overwrite this and all others
    force build.gradle
   create src/main/resources/config/bootstrap-heroku.yml
   create src/main/resources/config/application-heroku.yml

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 3m 29s
remote:        12 actionable tasks: 12 executed
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:        Done: 197.8M
remote: -----> Launching...
remote:        Released v7
remote:        https://health-by-points.herokuapp.com/ deployed to Heroku
...
remote: Verifying deploy... done.
To https://git.heroku.com/health-by-points.git
 * [new branch]      HEAD -> main
```

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.

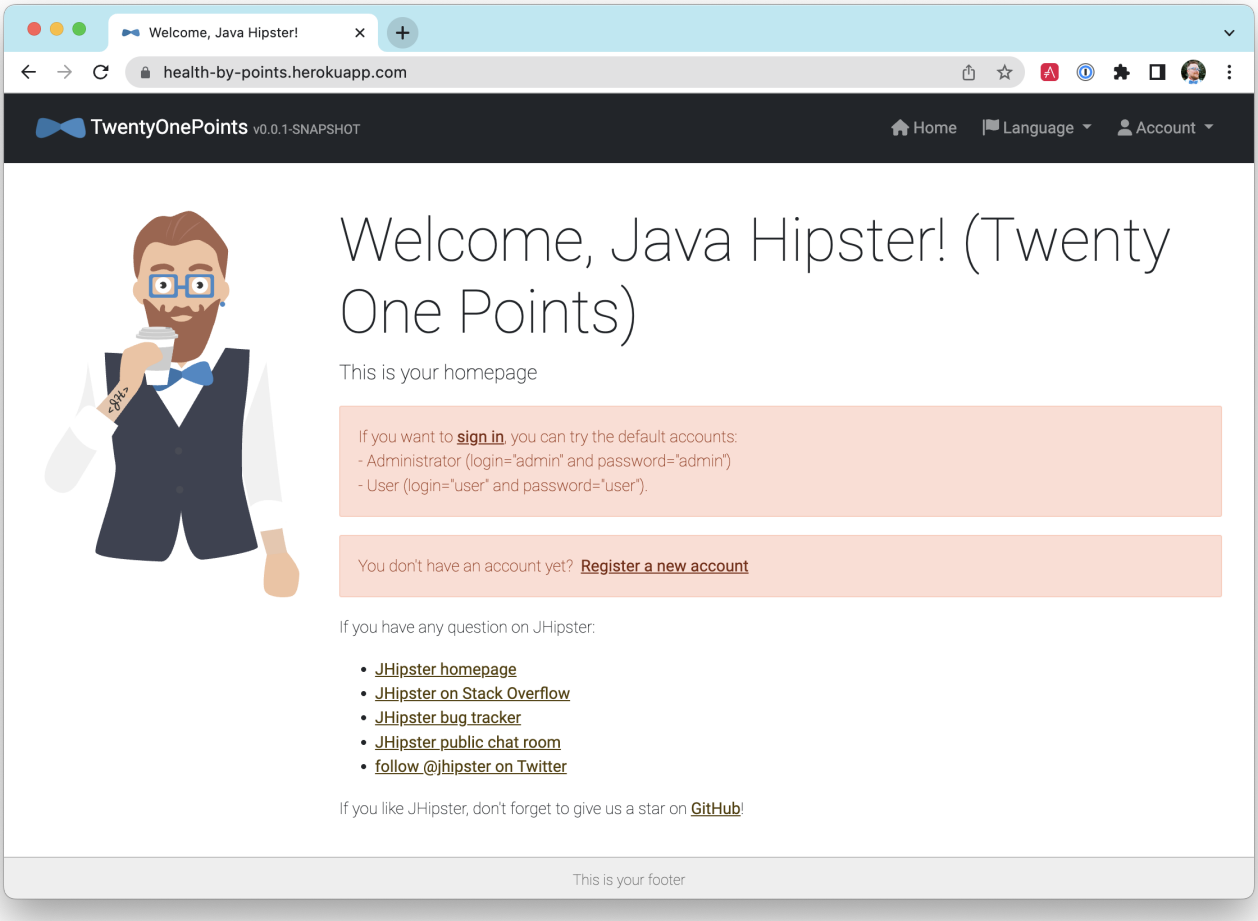


Figure 12. First deployment to Heroku

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 main`.

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 bootJar -Pprod
heroku jar:deploy build/libs/*.jar --app health-by-points
```

With Maven, the commands look slightly different:

```
./mvnw package -Pprod
heroku jar:deploy target/*.jar --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.

```
2022-11-08T05:17:22.489474+00:00 app[web.1]: 2022-11-08T05:17:22.488Z ERROR 4
  o.z.problem.spring.common.AdviceTraits   : Internal Server Error
2022-11-08T05:17:22.489494+00:00 app[web.1]:
org.springframework.web.util.NestedServletException:
  Handler dispatch failed; nested exception is java.lang.NoSuchFieldError:
INDEX_CONTENT_TYPE
```

I created an issue in the JHipster project saying that Elasticsearch doesn't work out of the box with Heroku. I contacted the Elastic team to determine the best solution. They recommended starting a free

trial on Elastic Cloud. After logging in, I created a deployment called **21-Points Health**. I used the default settings, selected **7.17.7** as the version, and pressed **Create deployment**.

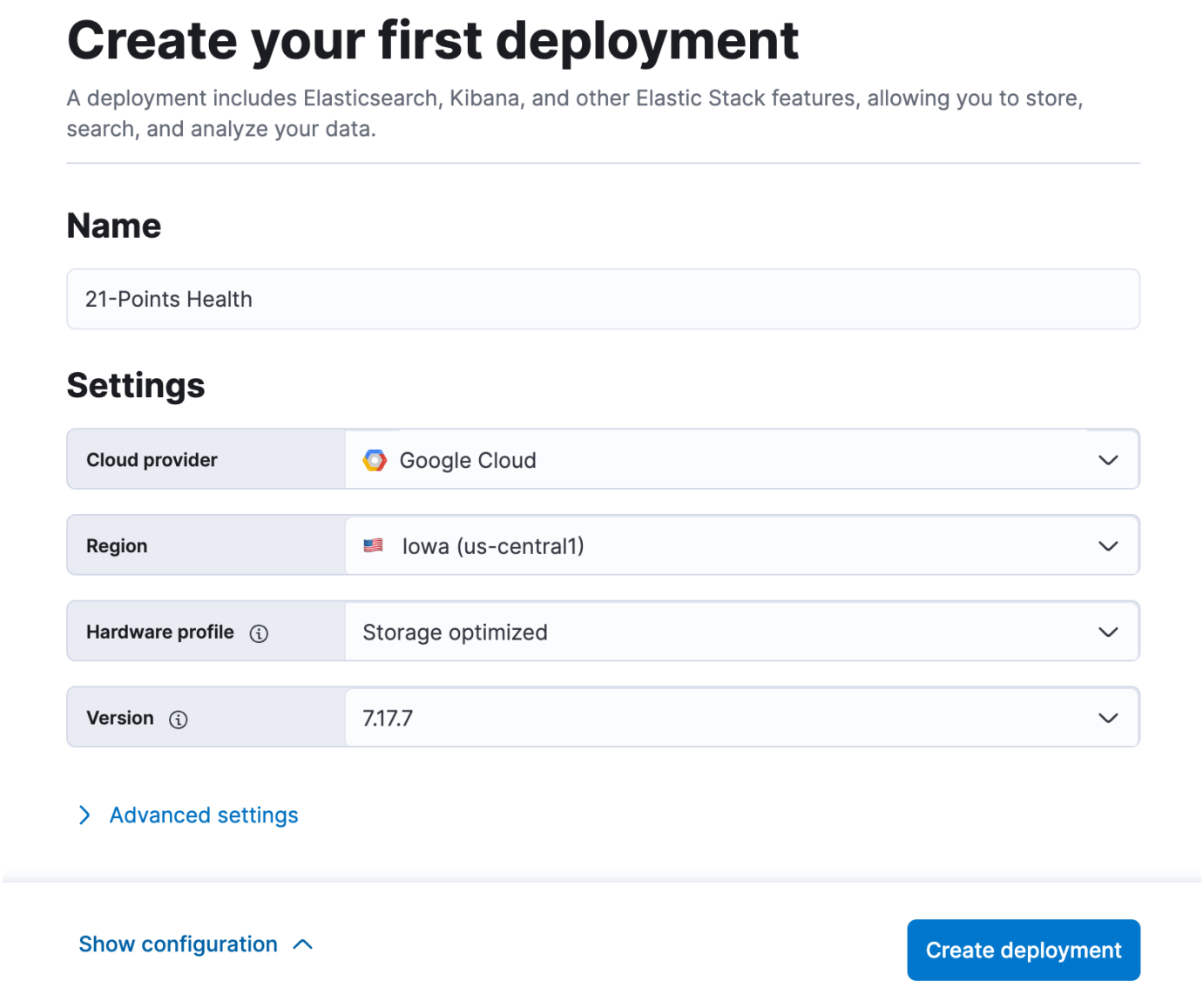


Figure 13. Elastic Cloud settings

I tried the latest version, but it resulted in an "Unable to parse response body" error.

I downloaded my credentials from the following screen and clicked **Continue**. Next, I selected **Manage this deployment** from the menu and copied the Elasticsearch endpoint.

I set the credentials and endpoint URL as a new `ELASTIC_URL` environment variable on Heroku.

```
heroku config:set ELASTIC_URL=https://elastic:<password>@<endpoint>
```

To fix my JHipster app so it recognized this variable, I modified `heroku.gradle` to remove the entire

block below for Bansai (that no longer works):

Listing 40. gradle/heroku.gradle

```
// force dependency version as used bonsai add-on as of now only supports 7.10.x
// https://github.com/jhipster/generator-jhipster/issues/18650
def bonsaiElasticSearchVersion = "7.10.2"
if (System.getenv("DYNO") != null) {
    configurations {
        all { ... }
    }
}
```

And I updated `application-heroku.yml` to use `ELASTIC_URL`.

Listing 41. src/main/resources/config/application-heroku.yml

```
spring:
  ...
  elasticsearch:
    uris: ${ELASTIC_URL}
```

I committed these changes and ran `git push heroku main` to redeploy the application.

Mail on Heroku

This time, when I tried to register, I received an error when my `MailService` tried to send me an activation e-mail.

```
2022-11-27T22:05:47.068322+00:00 app[web.1]: 2022-11-27T22:05:47.067Z  WARN 4 --- [e-points-task-2]
  org.jhipster.health.service.MailService  : Email could not be sent to user 'mraible@gmail.com'
2022-11-27T22:05:47.068339+00:00 app[web.1]:
2022-11-27T22:05:47.068341+00:00 app[web.1]: org.springframework.mail.MailSendException:
  Mail server connection failed; nested exception is com.sun.mail.util.MailConnectException:
  Couldn't connect to host, port: localhost, 25; timeout -1;
2022-11-27T22:05:47.068342+00:00 app[web.1]: nested exception is:
2022-11-27T22:05:47.068343+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 sendgrid on health-by-points... free
Created sendgrid-spherical-88389 as SENDGRID_PASSWORD, SENDGRID_USERNAME
```

```
Use heroku addons:docs sendgrid to view documentation
```

Then I updated `application-prod.yml` to use `SENDGRID_API_USER` and `SENDGRID_API_KEY` environment variables for mail, as well as to turn on authentication.

Listing 42. src/main/resources/config/application-prod.yml

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

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

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

The `SENDGRID_USERNAME` and `SENDGRID_PASSWORD` variables will not work to send email. You need to create an API Key instead. You can do this by navigating to your app in Heroku's dashboard. Then, select **Resources** > **Twilio SendGrid** and then the **Setup Guide** at the bottom. Create a sender identity that matches the `from` value above. Then, create an API key using the SMTP Relay integration.

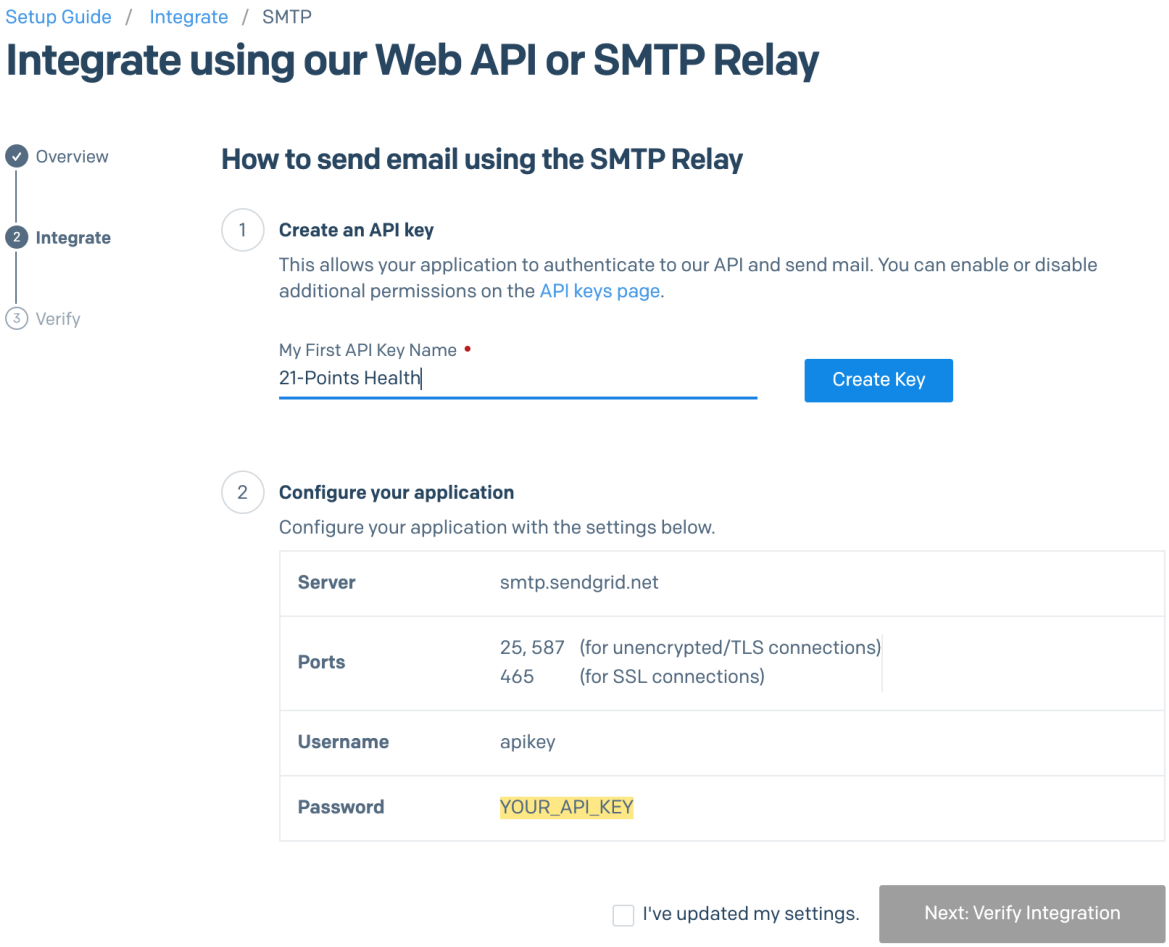


Figure 14. Create SendGrid API key

Once you have an API key, set the user and password for SendGrid on Heroku:

```
heroku config:set SENDGRID_API_USER=apikey SENDGRID_API_KEY=SG...
```

After redeploying, I logged in to my Heroku app with administrator credentials. I deleted the user I'd tried to add previously. I added the user again and smiled when I received the activation email.

Monitoring and analytics

JHipster generates the code necessary for Google Analytics in every application's `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`:

Listing 43. 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`). However, it's a bit different if you're using the heroku-deploy plugin.

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

```
./gradlew bootJar -Pheroku,prod
heroku buildpacks:clear
heroku jar:deploy build/libs/*.jar --includes newrelic-agent.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-agent.jar $JAVA_OPTS -Xmx256m -jar build/libs/*.jar ...
```

If you want to deploy using `git push heroku main` after using the heroku-deploy plugin, you'll have to run `heroku buildpacks:clear`.

To ensure `newrelic-agent.jar` is included when running `git push`, you'll need to modify your `.gitignore` to allow it and add it to Git with `git add newrelic.jar`.

```
*.jar
!newrelic-agent.jar
```

Securing user data

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

To fix this data leakage, I enhanced the Java code to allow only users that own an entity to edit it.

Here's some pseudocode to show the logic:

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

See 21-points#106 for all the changes that I needed to make in resource classes, search repositories, 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
...
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*
? *Heroku*: name of your Heroku Application ? health-by-points
   create Jenkinsfile
   create src/main/resources/idea.gdsl
    force .yo-rc-global.json
    force .yo-rc.json
   create src/main/docker/jenkins.yml
```

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

Jenkins Options

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

- Deploy artifact to an Artifactory.
- Analyze code with Sonar.
- Build and publish a Docker image.
- Add Snyk dependency scanning for security vulnerabilities.

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

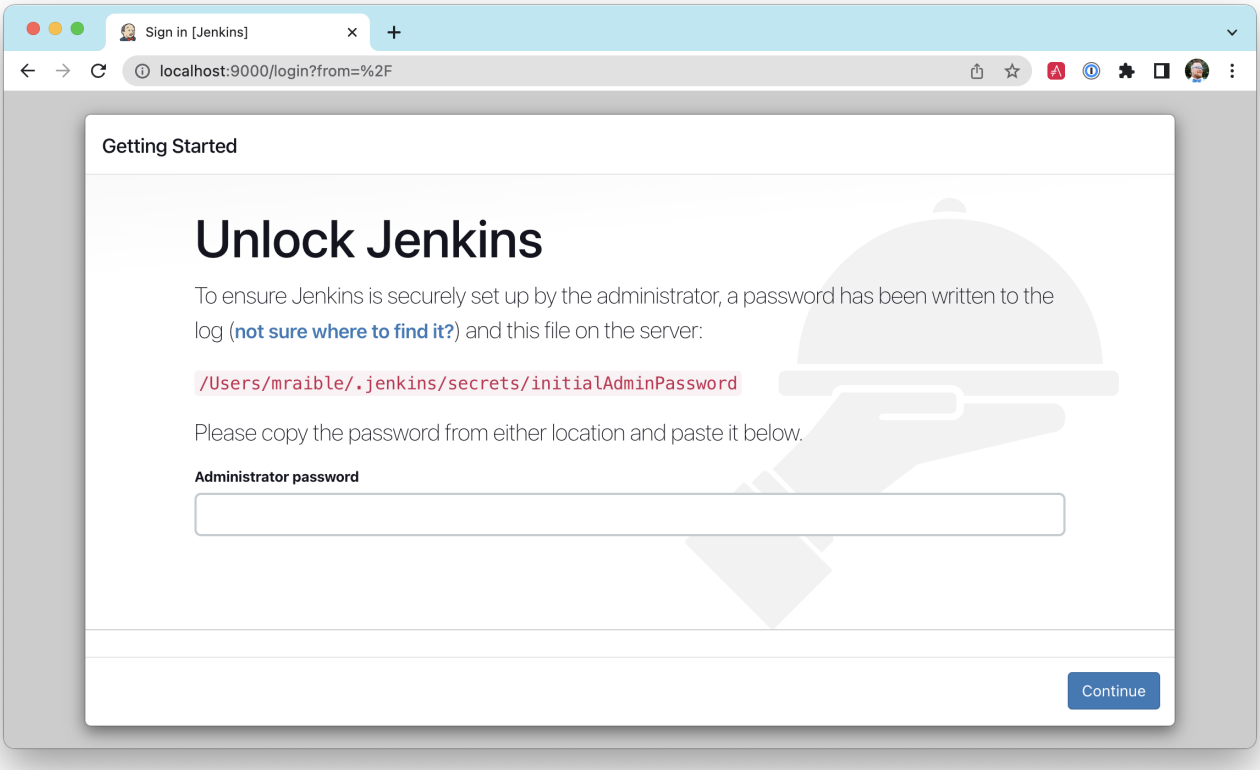


Figure 15. Unlock Jenkins

Next, I installed selected plugins and waited while everything completed downloading.

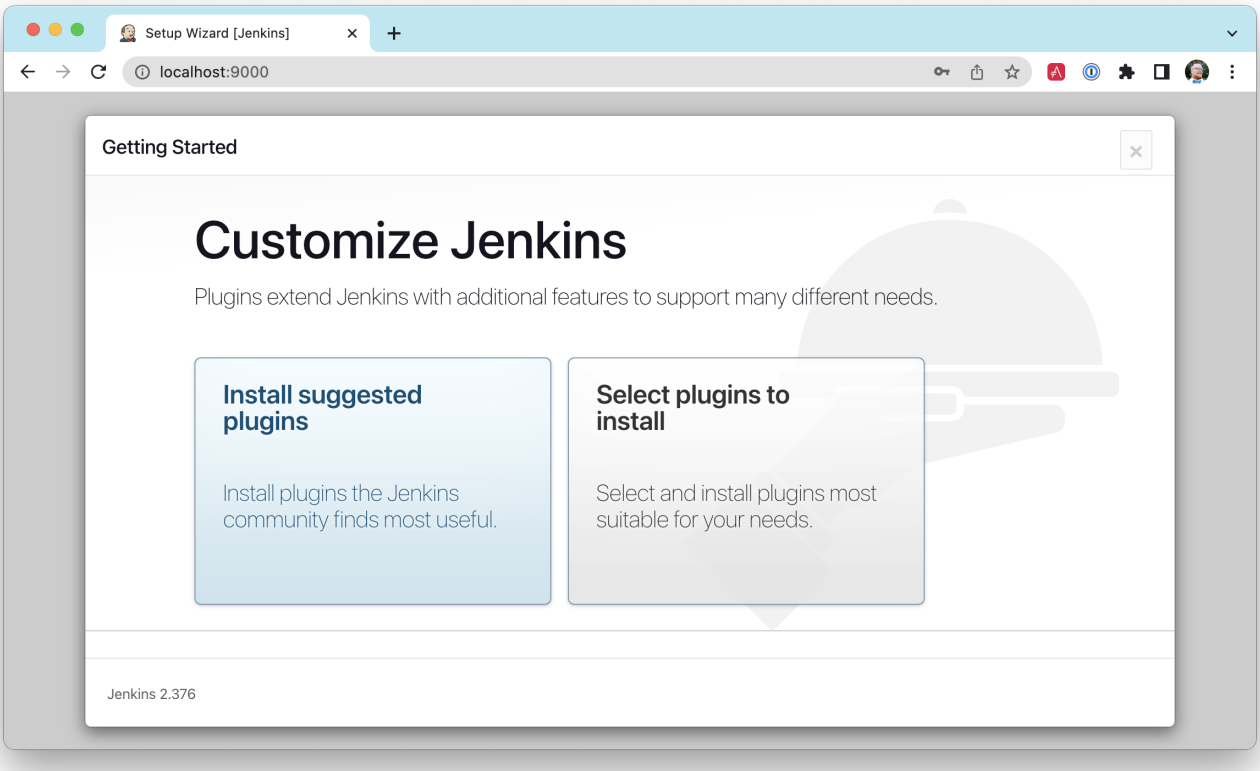


Figure 16. Customize Jenkins

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.

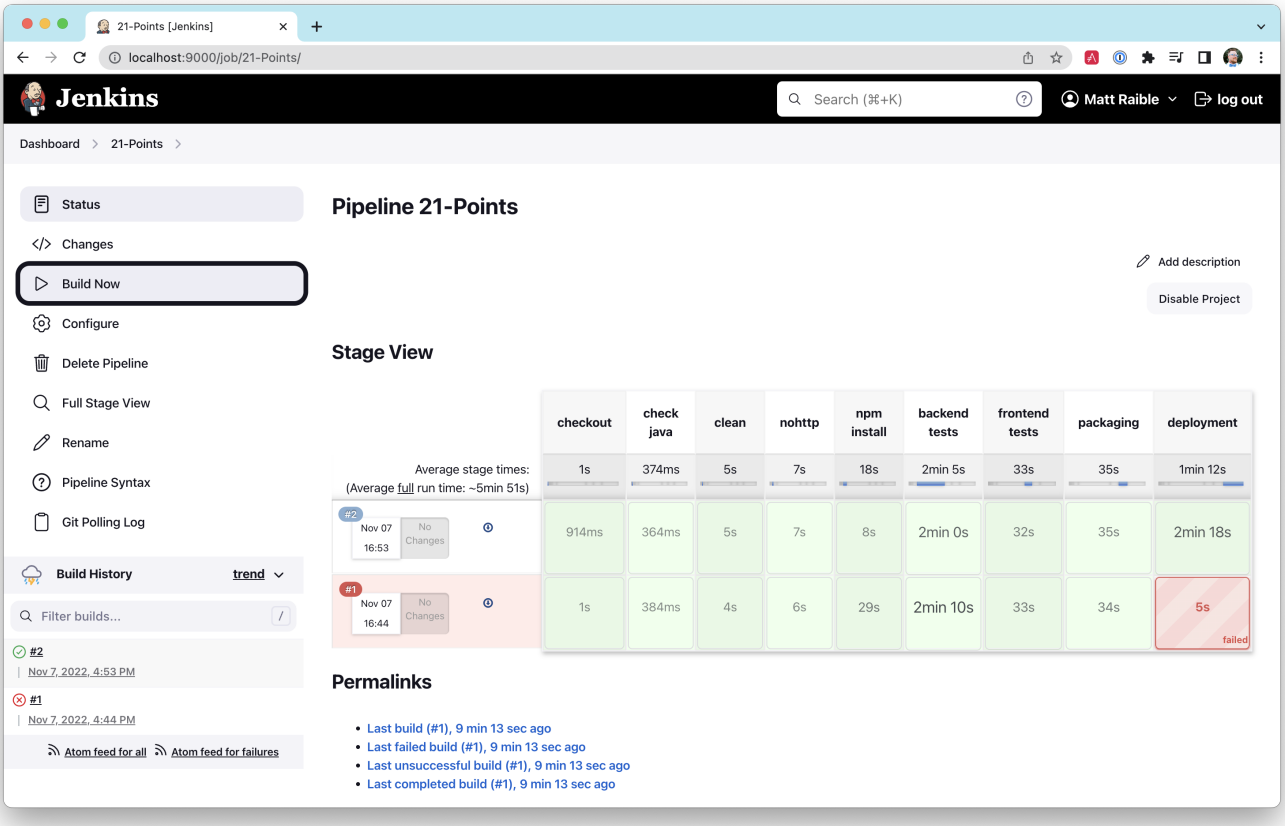


Figure 17. Jenkins build #1

It's possible the `deployment` stage will fail for you the first time (like it did for me above). If this happens, stop Jenkins, run `heroku login`, then restart Jenkins.

I modified `Jenkinsfile` to add an `e2e tests` stage to run all the Cypress tests. Before I checked it in, I started the app in one terminal and ran `npm run e2e` in another. I discovered the footer covered the bottom of the points edit form. I had to adjust `points.cy.ts` to force button clicks on lines 121 and 131.

Listing 44. src/test/javascript/cypress/e2e/entity/points.cy.ts

```
cy.get(entityCreateCancelButtonSelector).click({force: true});
```

I checked in my changes to trigger another build.

Listing 45. 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('nohttp') {
        sh "./gradlew checkstyleNohttp --no-daemon"
    }

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

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

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

    stage('e2e tests') {
        sh '''./gradlew &
        bootPid=$!
        sleep 30
        npm run e2e
        kill $bootPid
        '''
    }
```

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

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

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

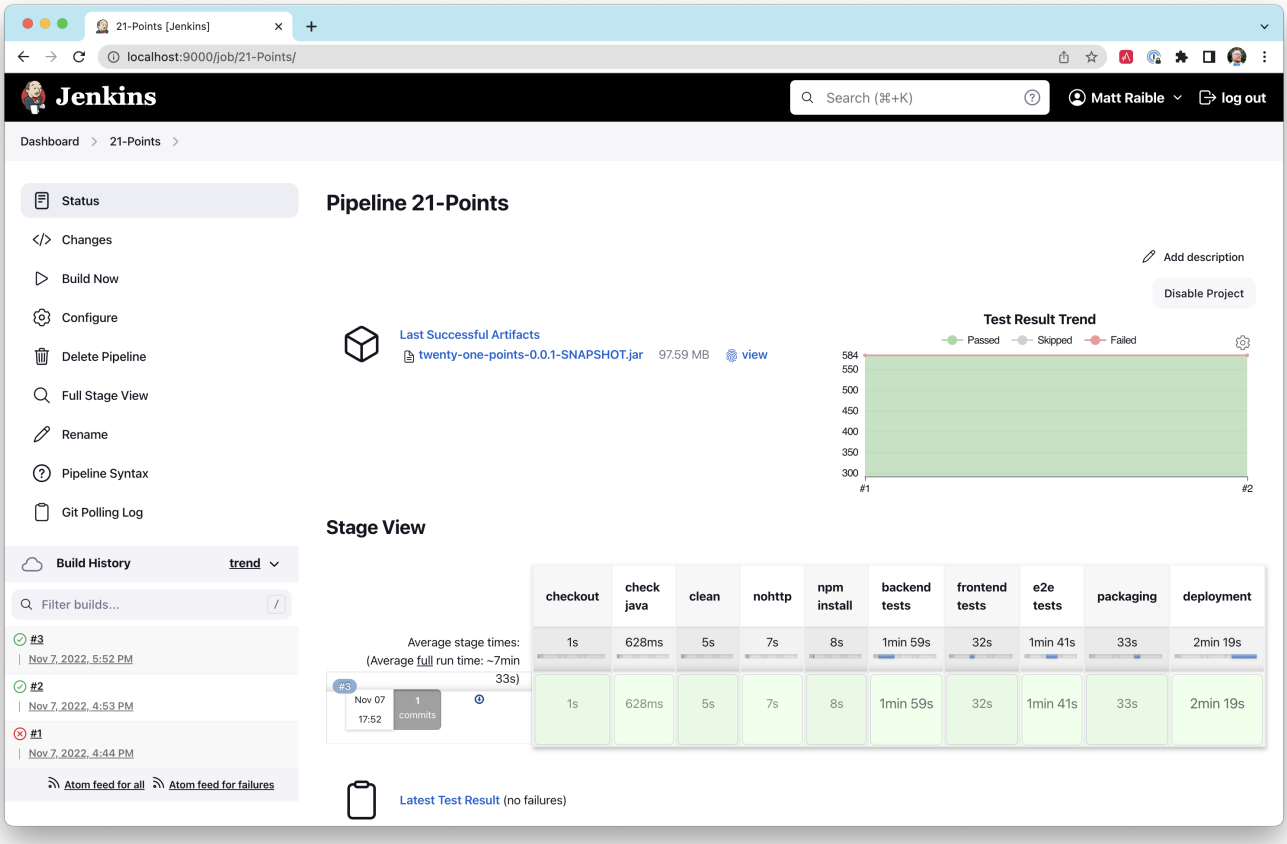


Figure 18. Jenkins success!

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 Sonar.

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

SonarQube 9.6.0 on Apple Silicon (M1)

SonarQube does not work with Apple Silicon using official images, a native image is not provided, and it fails in compatibility mode. You can build the Docker image locally to solve the problem:

```
git clone https://github.com/SonarSource/docker-sonarqube.git
cd docker-sonarqube/9/community
git checkout 9.6.0
docker build -t sonarqube:9.6.0-community .
```

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

```
./gradlew -Pprod clean check jacocoTestReport sonarqube
```

Once this process was 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?

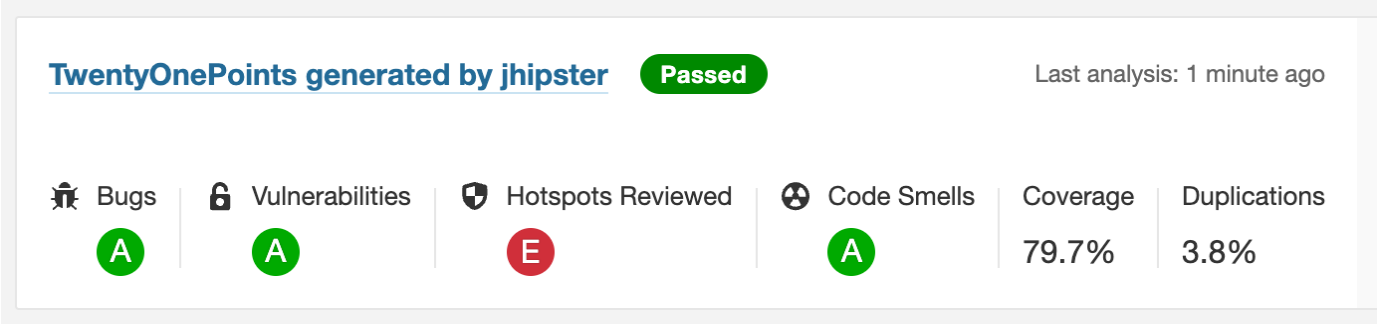


Figure 19. Sonar results

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 current 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 `npm run 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 `SecurityConfiguration.java` and added a rule to force a secure channel when an `X-Forwarded-Proto` header is sent.

Listing 46. src/main/java/org/jhipster/health/config/SecurityConfiguration.java

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        ...
    .and()
        .requiresChannel()
        .requestMatchers(r -> r.getHeader("X-Forwarded-Proto") != null)
        .requiresSecure();
}
```

JHipster ships with PWA support for Angular. It's turned off by default. One of the main components of a PWA is a service worker. To enable it, adjust the code in `src/main/webapp/app/app.module.ts`:

```
// Set this to true to enable service worker (PWA)
ServiceWorkerModule.register('ngsw-worker.js', { enabled: true }),
```

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 in Chrome to perform an analysis. You can see the results in the following image.

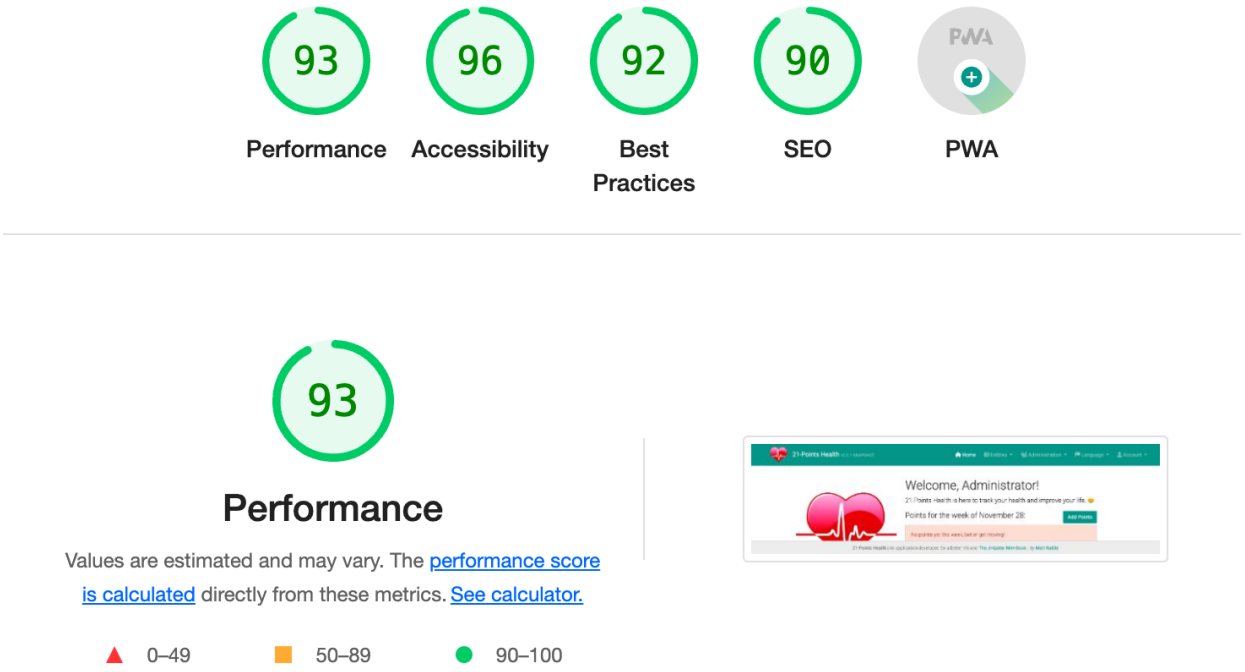


Figure 20. Lighthouse analysis

Source code

After getting this application into a good enough state, I pushed it 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.

Summary

This section showed 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, add methods to client services, and manipulate data to display pretty charts.

After modifying the application to look like my UI mockups, I showed you how to deploy it 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 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 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 three UI frameworks: Angular, React, and Vue. Since this is a mini-book, I'm going to stick with showing Angular only. You can see from the following graphs that React and Angular are the most popular JavaScript frameworks.

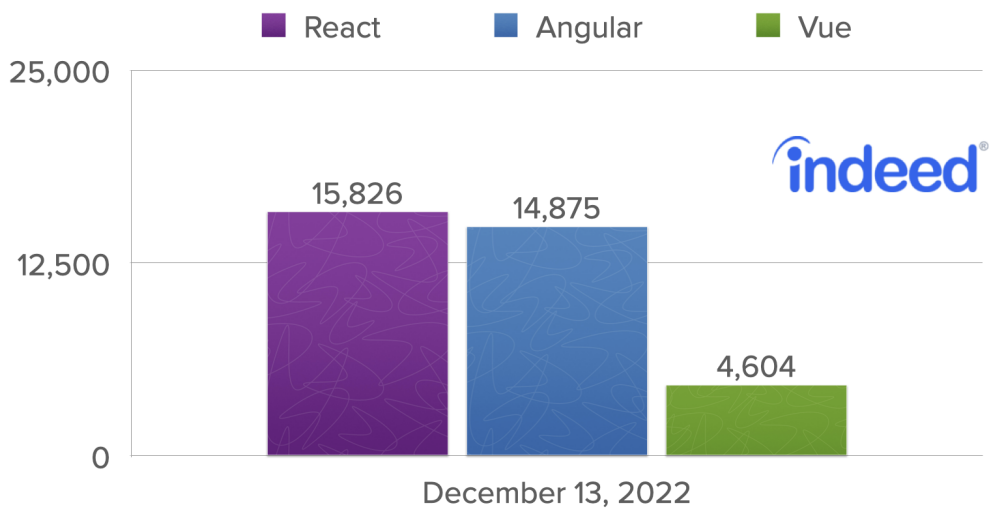


Figure 21. Jobs on Indeed, December 2022

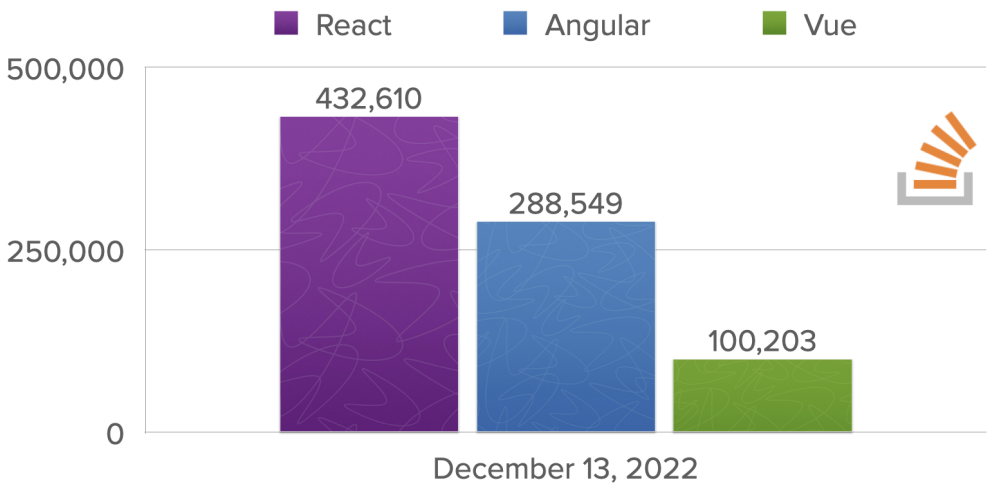


Figure 22. Stack Overflow Tags, December 2022

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 how 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 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. This syntax 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, announced 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 in September 2016. Angular 4.0 was released in March 2017. JHipster 4.6.0 was released on July 6, 2017, and was the first release to contain production-ready Angular support. JHipster 7 uses Angular 14. Angular released version 15 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 render this component 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 { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

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 you create or retrieve from a service. The view is an HTML template, and the component is a class that sets variables to be read by the template.

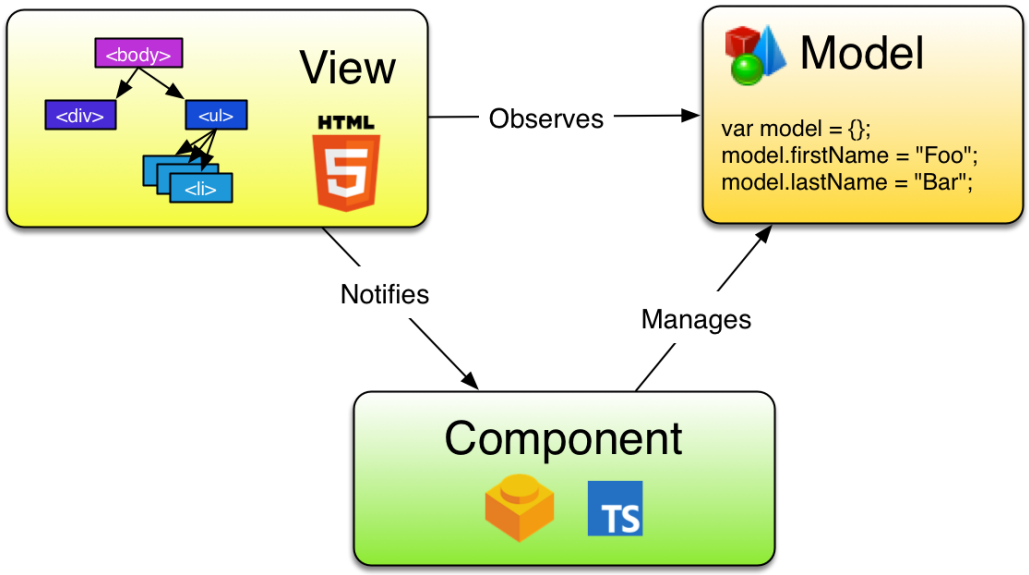


Figure 23. MVC in Angular

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

Listing 47. 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.

Listing 48. 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({
      next: (data: any) => {
        this.searchResults = data;
      },
      error: 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.

Listing 49. AppModule

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/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 app's main entry point, `AppComponent` in this case, you'll need to specify the router outlet in a template.

Listing 50. 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
      next: (data: any) => {
        this.searchResults = data;
      },
      error: 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 (357 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 (21 bytes)
CREATE src/app/search/search.component.spec.ts (599 bytes)
CREATE src/app/search/search.component.ts (202 bytes)
UPDATE src/app/app.module.ts (475 bytes)
```

Does your API return data like the following?

```
[
  {
    "id": 1,
    "name": "Nikola Jokić",
    "phone": "(720) 555-1212",
    "address": {
      "street": "2000 16th Street",
      "city": "Denver",
      "state": "CO",
      "zip": "80202"
    }
  },
  {
    "id": 2,
    "name": "Jamal Murray",
    "phone": "(303) 321-8765",
    "address": {
      "street": "2654 Washington Street",
      "city": "Lakewood",
      "state": "CO",
      "zip": "80568"
    }
  },
  {
    "id": 3,
    "name": "Aaron Gordon",
    "phone": "(303) 323-1233",
    "address": {
      "street": "46 Creekside Way",
      "city": "Winter Park",
      "state": "CO",
      "zip": "80482"
    }
  }
]
```

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

```
<table *ngIf="searchResults.length">
  <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 Bare Bones 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 many CSS classes and HTML structures that allow you to develop HTML user interfaces that look good. 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 that accommodates different screen sizes and 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 allow the container to alter its items' width/height (and order) to fill the available space best, 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 the viewport width. The table below shows how aspects of the grid system work across different devices.

	Extra small	Small	Medium	Large	Extra large	Extra extra large
Max container width	<576px	≥576px	≥768px	≥992px	≥1200px	≥1400px
Class prefix	`.col-`	`.col-sm-`	`.col-md-`	`.col-lg-`	`.col-xl-`	`.col-xxl-`

A basic example of the grid is shown below.

```
<div class="container text-center">
  <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>
</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 24. 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.

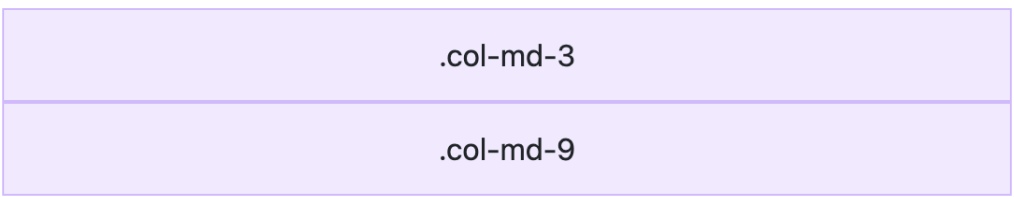


Figure 25. Basic grid on a mobile device

Bootstrap's grid can be used to align and position your application's elements, widgets, and features. Understanding a few basics is helpful if you 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="container text-center">
  <!-- Stack the columns on mobile by making one full-width and the other half-width -->
  <div class="row">
    <div class="col-md-8">.col-md-8</div>
    <div class="col-6 col-md-4">.col-6 .col-md-4</div>
  </div>

  <!-- Columns start at 50% wide on mobile and bump up to 33.3% wide on desktop -->
  <div class="row">
    <div class="col-6 col-md-4">.col-6 .col-md-4</div>
    <div class="col-6 col-md-4">.col-6 .col-md-4</div>
    <div class="col-6 col-md-4">.col-6 .col-md-4</div>
  </div>

  <!-- Columns are always 50% wide, on mobile and desktop -->
  <div class="row">
    <div class="col-6">.col-6</div>
    <div class="col-6">.col-6</div>
  </div>
</div>
```

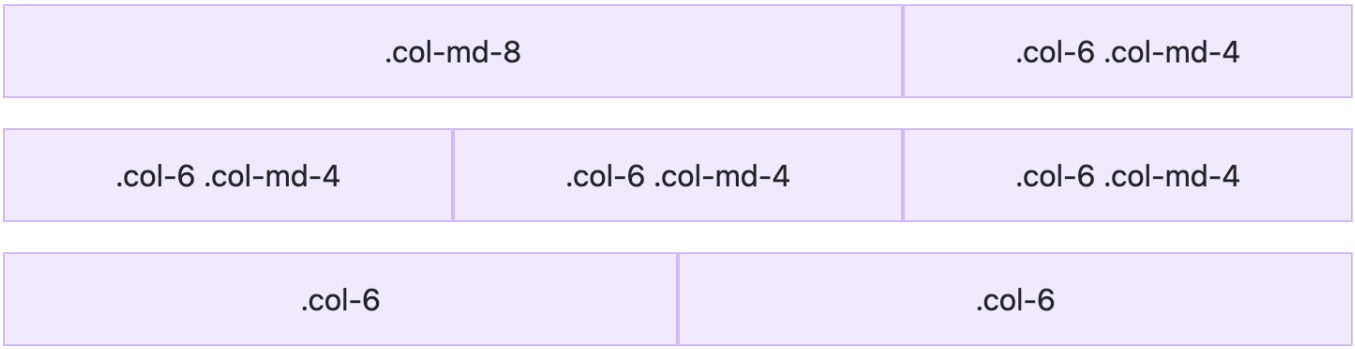


Figure 26. Advanced grid

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) { ... }

// Extra extra large devices (extra large desktops, 1400 px and up)
@media (min-width: 1400px) { ... }
```

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 several utility classes that can be used to show and hide elements based on the browser size, like `.d-[xs|sm|md|lg|xl|xxl]-block` and `.d-[xs|sm|md|lg|xl|xxl]-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`, `xl`, and `xxl`

The media queries affect screen widths with the given breakpoint or *larger*. For example, `.d-lg-none` sets `display: none` on both `lg`, `xl`, and `xxl` 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, 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="mb-3">
  <label for="description">Description</label>
  <textarea class="form-control" rows="4" name="description" id="description"></textarea>
</div>
```

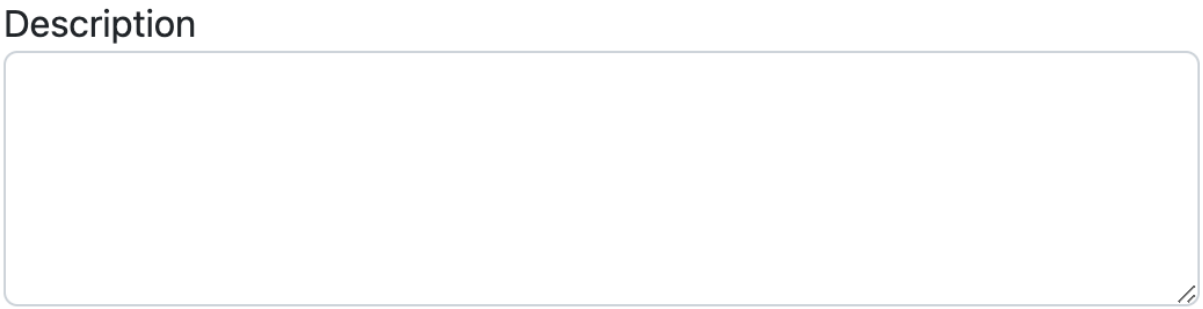


Figure 27. Basic form element

If you'd like to indicate that this form element is invalid, you'll need to modify the above HTML to display validation warnings.

```
<div class="mb-3">
  <label for="description">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>
```

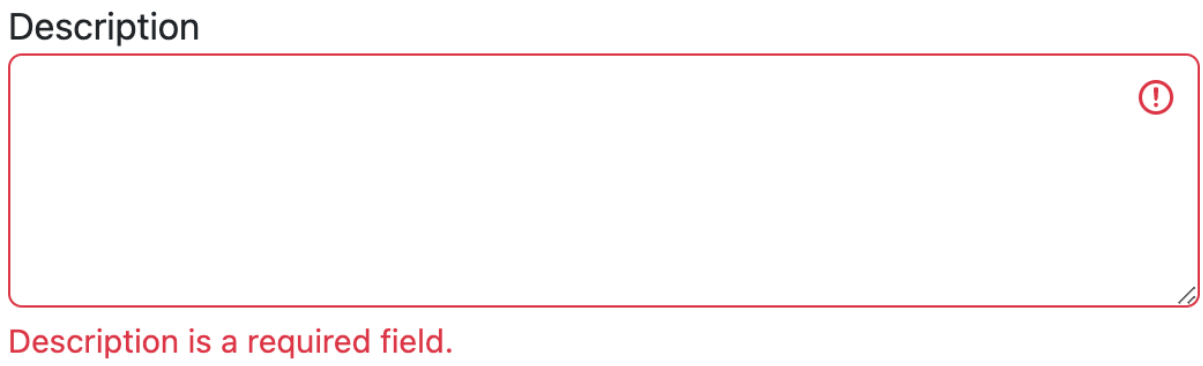


Figure 28. Form element with validation

CSS

When you add Bootstrap's CSS to an 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 several `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 29. Buttons

Components

Bootstrap ships with several components included. Some require JavaScript; some only require HTML5 markup and CSS classes. Its rich set of components has 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.

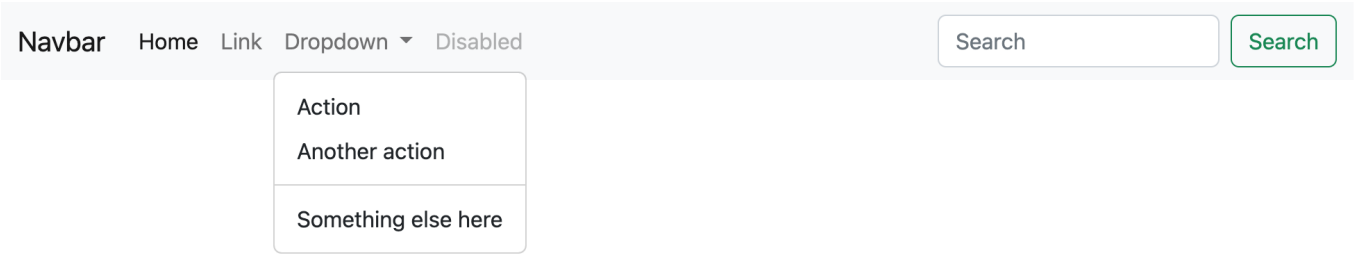


Figure 30. Navbar with dropdown

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

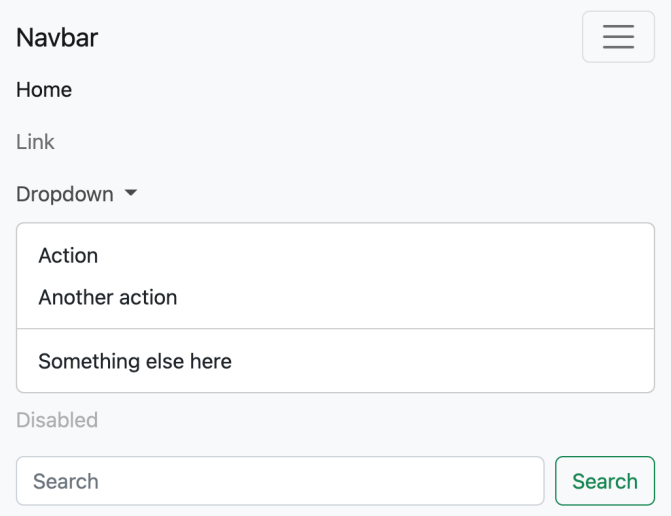


Figure 31. Navbar on mobile

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. An example without ARIA attributes below shows the basic structure.

```
<nav class="navbar navbar-expand-lg bg-light">
  <div class="container-fluid">
    <a class="navbar-brand" href="#">Navbar</a>
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
            data-bs-target="#navbarSupportedContent">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarSupportedContent">
      <ul class="navbar-nav me-auto mb-2 mb-lg-0">
        <li class="nav-item">
          <a class="nav-link active" 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="#" role="button"
             data-bs-toggle="dropdown">
            Dropdown
          </a>
          <ul class="dropdown-menu">
            <li><a class="dropdown-item" href="#">Action</a></li>
            <li><a class="dropdown-item" href="#">Another action</a></li>
            <li><hr class="dropdown-divider"></li>
            <li><a class="dropdown-item" href="#">Something else here</a></li>
          </ul>
```

```
        </li>
        <li class="nav-item">
          <a class="nav-link disabled">Disabled</a>
        </li>
      </ul>
      <form class="d-flex" role="search">
        <input class="form-control me-2" type="search" placeholder="Search">
        <button class="btn btn-outline-success" type="submit">Search</button>
      </form>
    </div>
  </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">
  A simple primary alert—check it out!
</div>
<div class="alert alert-secondary" role="alert">
  A simple secondary alert—check it out!
</div>
<div class="alert alert-success" role="alert">
  A simple success alert—check it out!
</div>
<div class="alert alert-danger" role="alert">
  A simple danger alert—check it out!
</div>
<div class="alert alert-warning" role="alert">
  A simple warning alert—check it out!
</div>
<div class="alert alert-info" role="alert">
  A simple info alert—check it out!
</div>
<div class="alert alert-light" role="alert">
  A simple light alert—check it out!
</div>
<div class="alert alert-dark" role="alert">
  A simple dark alert—check it out!
</div>
```

This renders alerts like the following.

Figure 32. 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="btn-close" data-bs-dismiss="alert" aria-label="Close"></button>
</div>
```

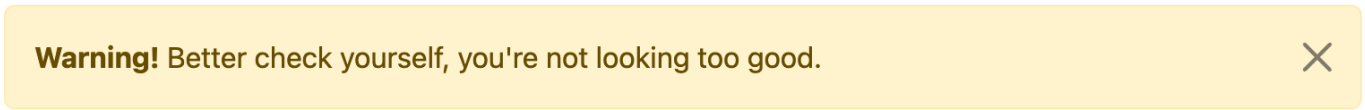


Figure 33. Closeable alert

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 contain symbols and glyphs instead of text. Because of their small size, you can style, scale, and load them quickly.

Bootstrap Icons is an icon library created by the Bootstrap team. You can install them using `npm i bootstrap-icons`. You can use them as SVGs, SVG sprites, or web fonts.

Font Awesome has 2,018 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 34. Buttons with icons

Customizing CSS

If you'd like to override Bootstrap classes in your project, just put the override rule in a CSS file that comes after Bootstrap's CSS. Or you can modify `src/main/webapp/content/scss/global.scss` directly. 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 the aforementioned `global.scss`.

Listing 51. src/main/webapp/scss/vendor.scss

```
/* after changing this file run 'npm run webapp: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 5 components powered by Angular instead of jQuery.

Popular alternatives to Bootstrap include Angular Material and Ionic Framework. There is no support for these frameworks at this time. Integrating them would require that all templates be 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 translating your application into multiple languages easy. It also provides a service that allows changing the language at runtime.

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 `jhipster languages`. As of December 2022, JHipster supports 47 languages. You can also add a new language. To set the default language, modify `translation.module.ts` and its `setDefaultLang()` setting.

Listing 52. src/main/webapp/app/shared/language/translation.module.ts

```
export class TranslationModule {
  constructor(...) {
    translateService.setDefaultLang('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 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 7 uses 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 smaller bundles to be loaded by the browser. Its code-splitting abilities make it possible to break up large JavaScript applications into small chunks to 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.

Listing 53. 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 containing files for different scenarios: dev mode with hot reload, testing, and production optimizations.

- `logo-jhipster.png` is the logo that shows in desktop alerts for build notifications.
- `environment.js` specifies the project's version, i18n hash, and server API URL.
- `webpack.custom.js` is the main configuration file that defines a `development` and a `production` mode. Dev mode enables hot-reloading with Browsersync and desktop notifications. HTML templates are converted to JavaScript, and source maps are created for production.

To learn more about webpack, I recommend visiting its getting started guide.

WebSockets

The WebSocket API (aka WebSockets) is 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 polling 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:

- RxStomp: STOMP stands for "simple text-oriented messaging protocol".
- SockJS: SockJS provides a WebSocket-like object. If native WebSockets are unavailable, they fall 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 posted to the `/websocket/tracker` endpoint.

Listing 54. src/main/webapp/app/admin/tracker/tracker.component.ts

```
import { Component, Inject, Vue } from 'vue-property-decorator';
import { Subscription } from 'rxjs';

import TrackerService from './tracker.service';

@Component
export default class JhiTrackerComponent extends Vue {

  public activities: any[] = [];
  private subscription?: Subscription;

  @Inject('trackerService') private trackerService: () => TrackerService;

  public mounted(): void {
    this.init();
  }

  public destroyed(): void {
    if (this.subscription) {
      this.subscription.unsubscribe();
      this.subscription = undefined;
    }
  }

  public init(): void {
    this.subscription = this.trackerService().subscribe(activity => {
      this.showActivity(activity);
    });
  }

  public showActivity(activity: any): void {
    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.splice(index, 1);
          this.activities.push(activity);
        }
      }
```

```
    }
    if (!existingActivity && activity.page !== 'logout') {
      this.activities.push(activity);
    }
  }
}
```

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

Listing 55. src/main/webapp/app/core/tracker/tracker.service.ts

```
import * as SockJS from 'sockjs-client';
import { Observer, map } from 'rxjs';
import VueRouter from 'vue-router';
import { Store } from 'vuex';
import { RxStomp } from '@stomp/rx-stomp';

import { AccountStateStorable } from '@/shared/config/store/account-store';

const DESTINATION_TRACKER = '/topic/tracker';
const DESTINATION_ACTIVITY = '/topic/activity';

export default class TrackerService {
  private rxStomp: RxStomp;

  constructor(
    private router: VueRouter,
    private store: Store<AccountStateStorable>,
  ) {
    this.stomp = new RxStomp();
    this.router.afterEach(() => this.sendActivity());

    this.store.watch(
      (_state, getters) => getters.authenticated,
      (value, oldValue) => {
        if (value === oldValue) return;
        if (value) {
          return this.connect();
        }
        return this.disconnect();
      }
    );
  }

  get stomp() {
```

```
    return this.rxStomp;
  }

  set stomp(rxStomp) {
    this.rxStomp = rxStomp;
    this.rxStomp.configure({
      debug: (msg: string): void => {
        console.log(new Date(), msg);
      },
    });
    this.rxStomp.connected$.subscribe(() => {
      this.sendActivity();
    });
  }

  private connect(): void {
    this.updateCredentials();
    return this.rxStomp.activate();
  }

  private disconnect(): Promise<void> {
    return this.rxStomp.deactivate();
  }

  private getAuthToken() {
    const authToken = localStorage.getItem('jhi-authenticationToken') ||
      sessionStorage.getItem('jhi-authenticationToken');
    return JSON.parse(authToken);
  }

  private buildUrl(): string {
    const loc = window.location;
    const baseHref = document.querySelector('base').getAttribute('href');
    const url = '//' + loc.host + baseHref + 'websocket/tracker';
    const authToken = this.getAuthToken();
    if (authToken) {
      return `${url}?access_token=${authToken}`;
    }
    return url;
  }

  private updateCredentials(): void {
    this.rxStomp.configure({
      webSocketFactory: () => {
        return new SockJS(this.buildUrl());
      },
    });
```

```
  }

  private sendActivity(): void {
    this.rxStomp.publish({
      destination: DESTINATION_ACTIVITY,
      body: JSON.stringify({page: this.router.currentRoute.fullPath}),
    });
  }

  public subscribe(observer) {
    return this.rxStomp
      .watch(DESTINATION_TRACKER)
      .pipe(map(imessage => JSON.parse(imessage.body)))
      .subscribe(observer);
  }
};
```

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 following 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 both windows scroll in Chrome and in Safari. 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 many 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 running on connected devices.

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

```
{
  "name": "jhipster-book",
  "version": "7.0.0",
  "description": "The JHipster Mini-Book",
  "author": "Matt Raible <matt@raibledesigns.com>",
  "license": "Apache-2.0",
  "repository": {
    "type": "git",
    "url": "git@github.com:mraible/jhipster-book.git"
  },
  "devDependencies": {
    "gulp": "4.0.2",
    "browser-sync": "2.27.10"
  }
}
```

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

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

gulp.task('serve', function () {
  browserSync.init({
    server: '.'
  });

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

gulp.task('default', gulp.series(['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's 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 prevalent 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 this section showed you how you could use webpack. Finally, it showed you how WebSockets work and described the beauty of Browsersync.

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

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 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. Roy Fielding initially proposed REST 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 RESTful 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 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.

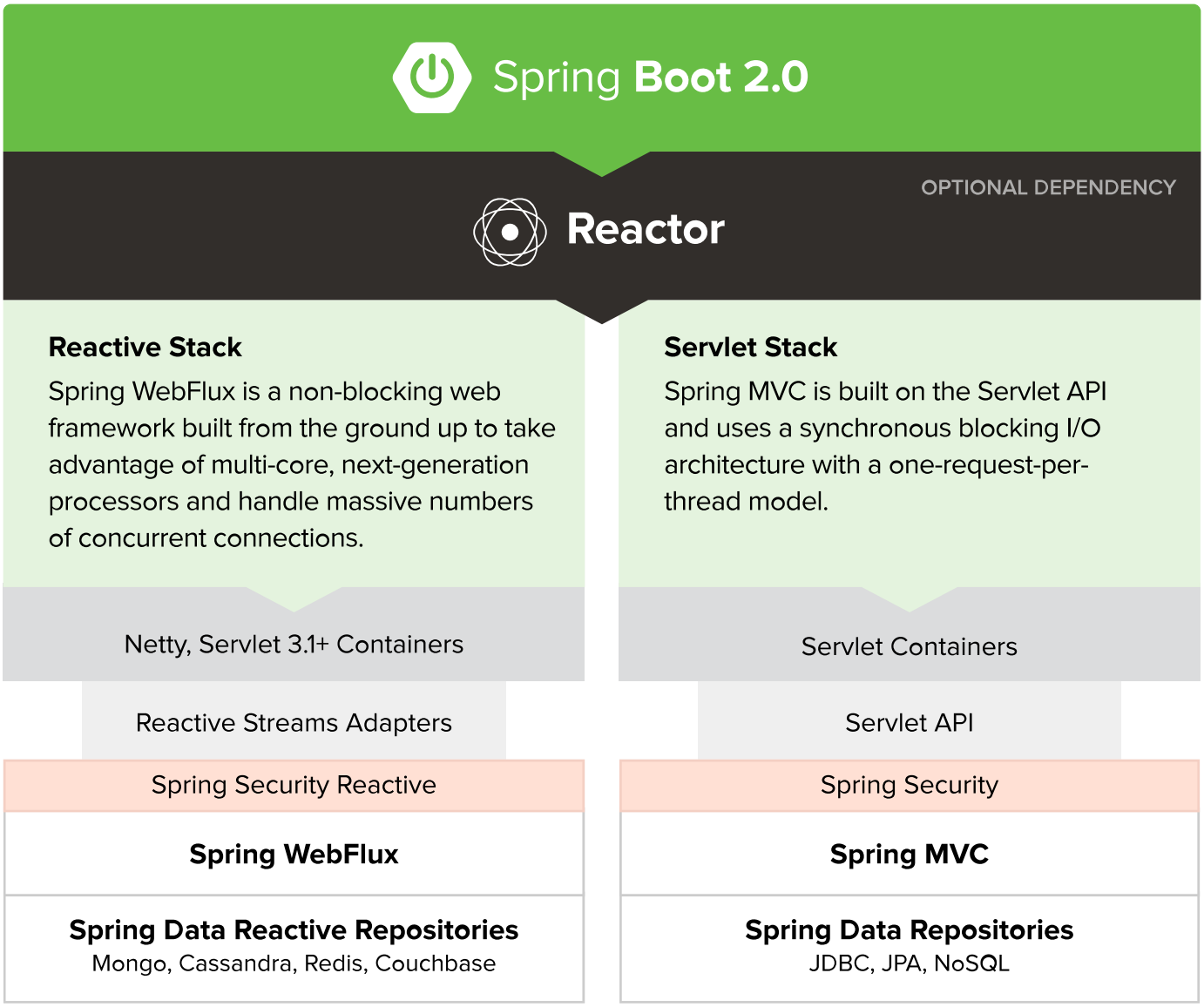


Figure 35. Spring Boot 2.0

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
- 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 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`.

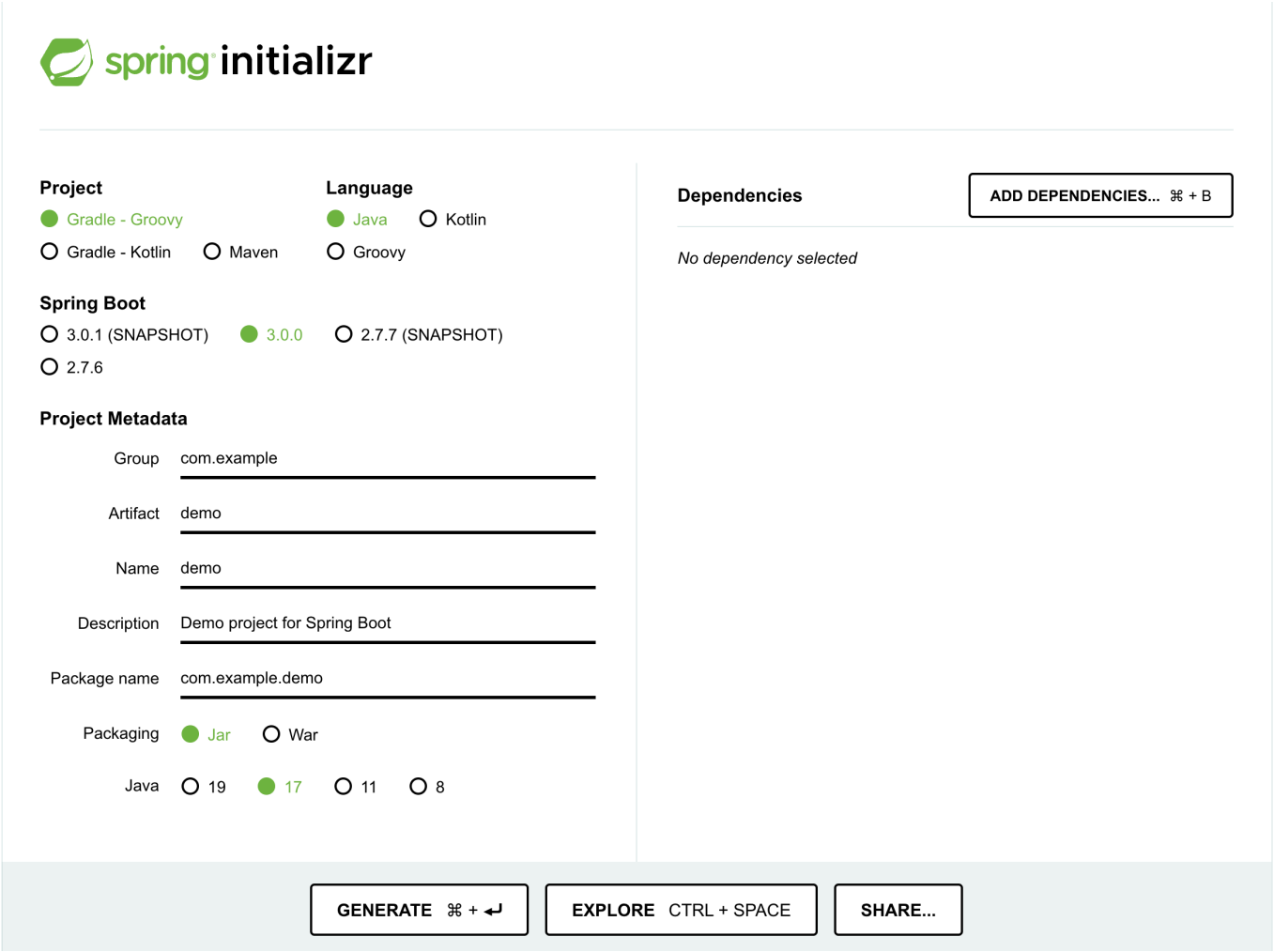


Figure 36. Spring Initializr in a browser

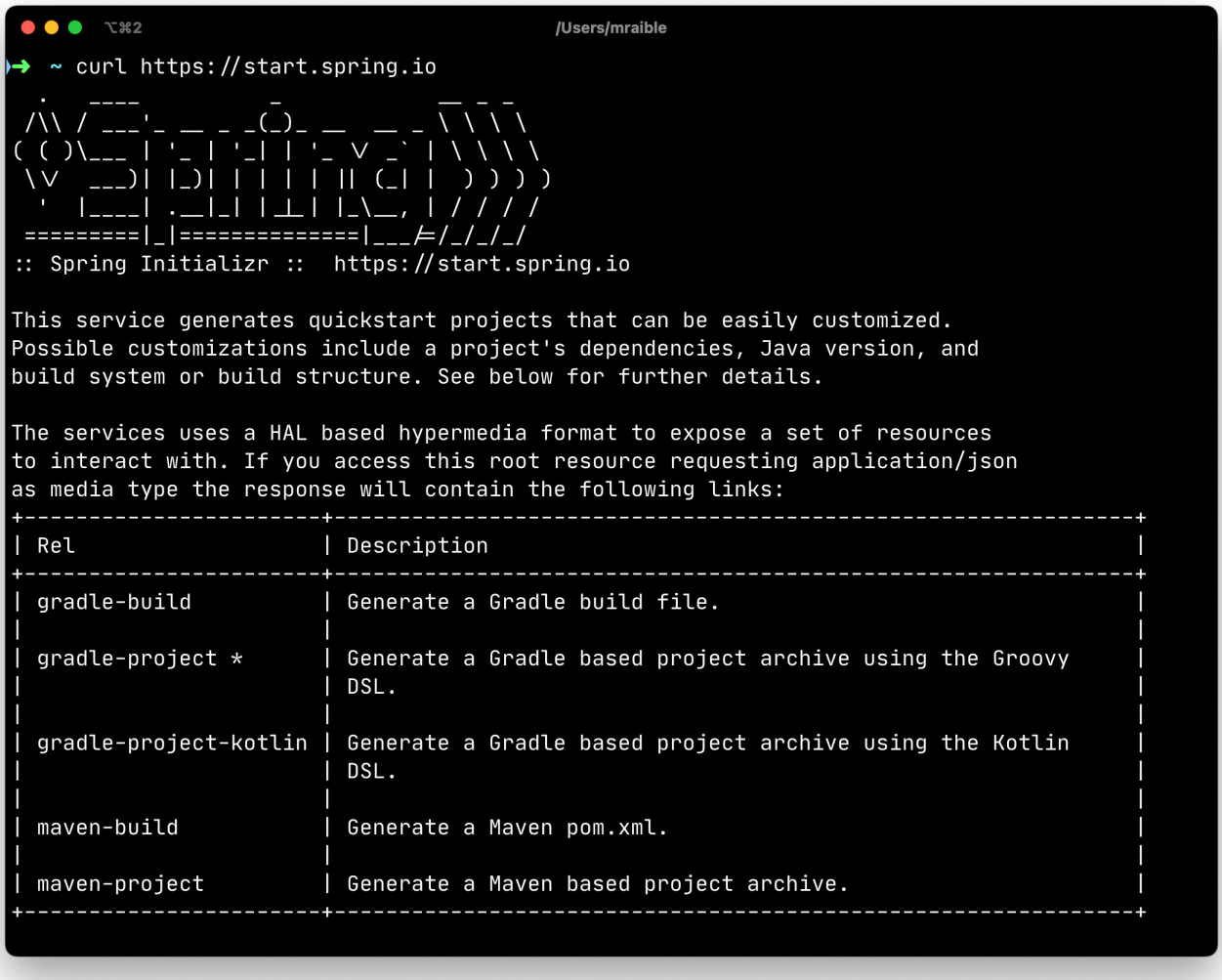


Figure 37. Spring Initializr via `curl`

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, you can create it with the following command.

```
spring init -d=web
```

This will create a Spring Boot project with the Spring Web dependency. Expand the `demo.zip` it downloads and add a `HelloController` in the same package as `DemoApplication`.

Listing 56. HelloController.java

```
package com.example.demo;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

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

To compile and run this application, simply type:

```
./gradlew bootRun
```

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 see 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.

The Spring CLI's command for creating this same app is `spring init -d=web,jpa,h2,actuator`.

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

```
.
├── HELP.md
├── build.gradle
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── settings.gradle
└── src
    ├── main
    │   ├── java
    │   │   └── com
    │   │       └── example
    │   │           └── demo
    │   │               └── DemoApplication.java
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        └── java
            └── com
                └── example
                    └── demo
                        └── DemoApplicationTests.java

16 directories, 10 files
```

`DemoApplication.java` is the heart of this application; the file and class name are irrelevant. What is relevant is the `@SpringBootApplication` annotation and the class's `public static void main` method, colloquially known as the main method.

Listing 57. 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 below the `DemoApplication` class.

Listing 58. src/main/java/demo/com/example/demo/DemoApplication.java

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.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 creating a CRUD repository for an entity easy. It automatically creates the implementation that talks to the underlying data store.

Listing 59. 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 found by calling its `findAll()` method. `CommandLineRunner` is an interface that indicates that a bean should run when it is contained within a `SpringApplication`.

Listing 60. 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());
    }
}
```

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

Listing 61. src/main/resources/data.sql

```
insert into blog (id, name) values (1, 'First');
```

```
insert into blog (id, name) values (2, 'Second');
```

And add the following property to `src/main/resources/application.properties`:

```
spring.jpa.defer-datasource-initialization=true
```

Start your application with `gradle bootRun` (or right-click → "Run in your IDE"), and you should see this default data show up in your logs.

```
INFO 65838 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
  : Tomcat started on port(s): 8080 (http) with context path ''
INFO 65838 --- [           main] com.example.demo.DemoApplication
  : Started DemoApplication in 1.742 seconds (process running for 1.913)
[Blog{id=1, name='First'}, Blog{id=2, name='Second'}]
```

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

Listing 62. 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 to 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 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 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
4. the classpath root

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

You can find more information about Spring Boot's external-configuration feature 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. You might load these files with the following:

`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 third-party library maintainer 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`.

Listing 63. spring-boot-actuator-autoconfigure-2.7.3.RELEASE.jar!/META-INF/spring.factories

```
org.springframework.boot.diagnostics.FailureAnalyzer=\
org.springframework.boot.actuate.autoconfigure.metrics.ValidationFailureAnalyzer
```

These configuration classes 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 give Spring Boot immense power and make it easy to use, configure, and override.

Actuator

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

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

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

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

Actuator's main features are endpoints, metrics, auditing, and process monitoring. Actuator auto-creates several 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.
- `/caches`—Exposes available caches. This endpoint is only available when the application context contains a `CacheManager`.
- `/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. This endpoint is only available when Flyway is on the classpath.
- `/health`—Returns information about application health.
- `/httptrace`—Returns trace information (by default, the last 100 HTTP requests). Requires an `HttpTraceRepository` bean.
- `/info`—Returns basic application info.
- `/integrationgraph`—Shows the Spring Integration graph when Spring Integration is on the classpath.
- `/loggers`—Shows and modifies the configuration of loggers in the application.
- `/liquibase`—Shows any Liquibase database migrations that have been applied. This endpoint is only available when Liquibase is on the classpath.
- `/metrics`—Returns performance information for the current application.
- `/mappings`—Returns a list of all `@RequestMapping` paths.
- `/quartz`—Shows information about Quartz Scheduler jobs.
- `/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).
- `/startup`—Shows the startup steps data collected by `ApplicationStartup`. Requires the application to be configured with a `BufferingApplicationStartup` bean.
- `/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-actuator
spring-boot-starter-data-jpa
spring-boot-starter-data-elasticsearch
spring-boot-starter-logging
spring-boot-starter-mail
spring-boot-starter-security
spring-boot-starter-thymeleaf
spring-boot-starter-web
spring-boot-starter-test
```

```
spring-boot-starter-undertow
```

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.

In the next chapter on microservices, I'll show you how to use Spring WebFlux in a reactive microservices architecture. If you'd like to learn how to use WebFlux in a monolith, I'd suggest reading Josh Long and my Build Reactive APIs with Spring WebFlux blog post.

Maven versus Gradle

Maven and Gradle are the 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 626 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 anything you want to implement in your build script. Out of the box, however, it will only build something if 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 and bundle it into a JAR file.

I've used both tools for building projects, and they both worked well. Maven works for me, but I've used it for almost 20 years and recognize that my history and experience might contribute to my bias toward it.

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.

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. 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 Settings > Build, Execution, Deployment > Compiler: enable `Build project automatically`
- Go to Advanced Settings and enable `Allow auto-make to start even if developed application is currently running`

Eclipse is a free alternative to IntelliJ IDEA. Its error highlighting (via auto-compile), code assist, and

refactoring support are excellent. When I started using it 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 needs better 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/TypeScript 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 the 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`. To use a plain-text value, you must prefix it with `{noop}`. See Spring Security's password storage documentation for more information.

The most basic Spring Security Java configuration creates a servlet `Filter` 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.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

@EnableWebSecurity
@Configuration
public class SecurityConfiguration {
```

```
    @Bean
    public PasswordEncoder passwordEncoder() {
       return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

    @Bean
    public UserDetailsService userDetailsService(PasswordEncoder passwordEncoder) {
        InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();
        manager.createUser(User.withUsername("user")
            .password(passwordEncoder.encode("password"))
            .roles("USER").build());
        return manager;
    }
}
```

There's not much code, but it provides many features:

- It requires authentication for 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
  - X-Frame-Options integration to help prevent clickjacking
- It integrates with HttpServletRequest API methods of: `getRemoteUser()`, `getUserPrinciple()`, `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 (created in 2006). OAuth 2.0 simplifies 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 the 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 several 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.

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, PostgreSQL, MySQL, MariaDB, Oracle, MSSQL)
  MongoDB
  Cassandra
  [BETA] Couchbase
  [BETA] Neo4j
  No database
```

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 only works with SQL databases (but not Oracle or Microsoft SQL Server).

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 many 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 Mongock 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.

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.

Listing 64. src/main/resources/config/liquibase/changelog/00000000000000_initial_schema.xml

```
<loadData
          file="config/liquibase/data/user.csv"
          separator=";"
          tableName="jhi_user"
          usePreparedStatements="true">
    <column name="id" type="numeric"/>
    <column name="activated" type="boolean"/>
    <column name="created_date" type="timestamp"/>
```

```
</loadData>
<dropDefaultValue tableName="jhi_user" columnName="created_date" columnDataType="${datetimeType}"/>
```

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 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 liquibaseDiffChangelog -PrunList=diffLog`.

Elasticsearch

Elasticsearch adds searchability to your entities. JHipster's Elasticsearch support requires using a SQL database. Spring Boot uses and configures Spring Data Elasticsearch. JHipster's entity sub-generator automatically indexes the entity and creates an endpoint to support searching its properties. Search superpowers are also added to the Angular UI so that you can search in your entity's list screen.

When using the (default) "dev" profile, you need to use an external Elasticsearch instance. The easiest way to run an external Elasticsearch instance is to use the provided Docker Compose configuration:

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

By default, the `SPRING_ELASTICSEARCH_URIS` property is set to talk to this instance in `application-dev.yml` and `application-prod.yml`:

```
spring:
  ...
  elasticsearch:
    uris: http://localhost:9200
```

You can override this setting by modifying these files, or using an environment variable:

```
export SPRING_ELASTICSEARCH_URIS=https://cloud-instance
```

Elasticsearch is used by some 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—"Accelerate results with Elastic across any cloud".
- Logstash—"Centralize, transform & stash your data".
- Kibana—"Your window into the Elastic Stack".

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 you can run a Java program. 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`.

To build your JHipster app with the production profile, use the preconfigured "prod" Maven profile.

```
mvn package -Pprod
```

With Gradle, it's:

```
gradle bootJar -Pprod
```

The "prod" profile will trigger a `webapp: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 reference your versioned, combined, and minified files.

You can deploy a JHipster application to your own JVM, Heroku, Kubernetes, and AWS.

I've deployed JHipster applications to Heroku and several cloud providers with Kubernetes, including Google Cloud, AWS, Azure, and Digital Ocean.

Summary

Spring Framework has one of the best track records for hipness in Javaland. It's remained backward compatible between many releases and has been an open-source project for over 20 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 shown you its many features, which you can enable with only a few lines of code. JHipster supports relational and NoSQL databases, allowing you to choose how you want your data stored. When creating a new application, you can select JPA, MongoDB, Cassandra, Couchbase, or Neo4j.

Liquibase will create your database schema and help you update your database when needed. Its diff feature provides an easy-to-use workflow for adding new properties to your JHipster-generated entities.

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 you can run Java. 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 microservices 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 break up your project into components that can function independently. 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 quickly.

Before we dive into the code tutorial, I'd like to talk 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 then, described this architecture as "fine-grained SOA". Martin Fowler and James Lewis wrote an article titled "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, etc. 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.

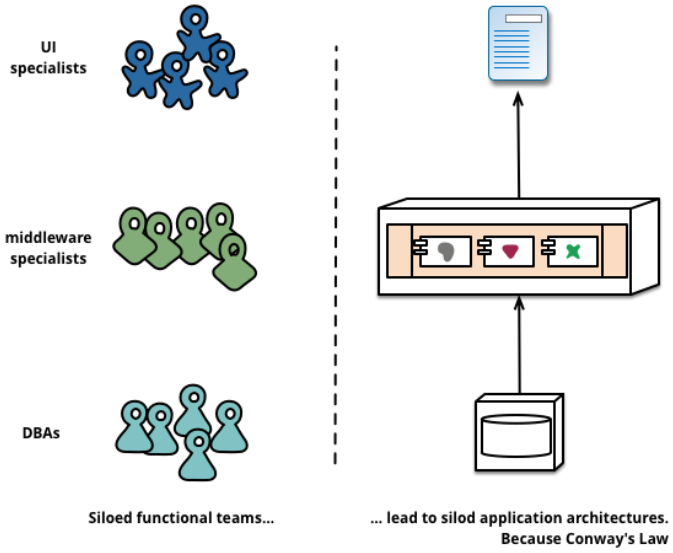


Figure 38. Conway's law

> 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 essentially equals 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
- 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), you can write applications 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 unable to create your components cleanly, you’re just shifting the 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

Reactive Java microservices

Spring Boot 2.0 introduced a new web framework called Spring WebFlux. Previous versions of Spring Boot only shipped with Spring MVC as an option. WebFlux offers a way for developers to do *reactive programming*. This means you can write your code with familiar syntax, and, as a result, your app will use fewer resources and scale better.

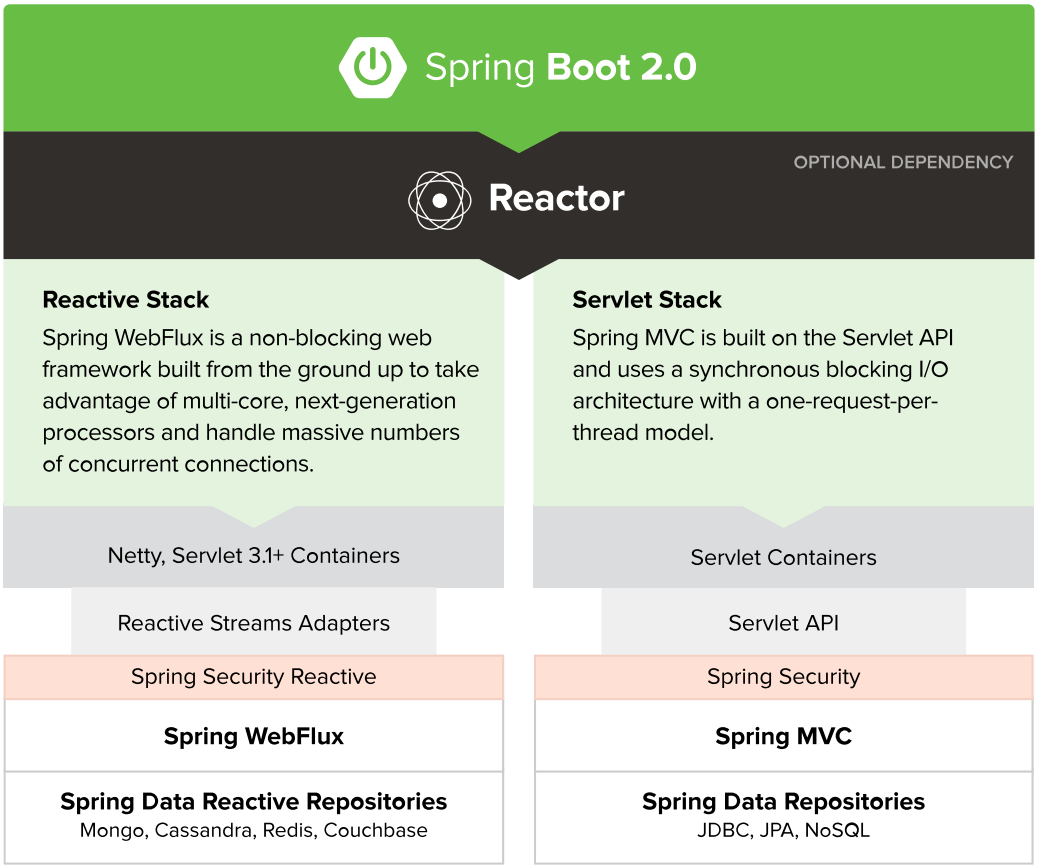


Figure 39. Spring Boot 2.0

Reactive programming isn't for every app. The general rule of thumb is it won't help you if you have < 500 requests/second. Chances are Spring MVC will perform as well as Spring WebFlux up to that point. When your traffic takes off, or if you need to process things faster than 500 requests/second, you should look at Spring WebFlux.

JHipster 7 introduced support for Spring WebFlux. This means you can generate a reactive microservices architecture with Spring Cloud Gateway and Spring Boot quickly and easily. This is a great way to get started with reactive programming.

Spring WebFlux's API has a similar syntax to Spring MVC. For example, here's the Spring MVC code for creating a new `Points` entity in a JHipster app created with `jhipster jdl 21-points.jh`.

```
@PostMapping("/points")
public ResponseEntity<Points> createPoints(@Valid @RequestBody Points points) throws URISyntaxException {
    log.debug("REST request to save Points : {}", points);
    if (points.getId() != null) {
        throw new BadRequestAlertException("A new points cannot already have an ID", ENTITY_NAME, "idexists");
    }
    Points result = pointsRepository.save(points);
    pointsSearchRepository.index(result);
    return ResponseEntity
        .created(new URI("/api/points/" + result.getId()))
        .headers(HeaderUtil.createEntityCreationAlert(applicationName, true, ENTITY_NAME, result.getId().toString()))
        .body(result);
}
```

The same functionality when implemented with Spring WebFlux returns a `Mono` and uses a more functional, streaming style to avoid blocking.

```
@PostMapping("/points")
public Mono<ResponseEntity<Points>> createPoints(@Valid @RequestBody Points points) throws URISyntaxException {
    log.debug("REST request to save Points : {}", points);
    if (points.getId() != null) {
        throw new BadRequestAlertException("A new points cannot already have an ID", ENTITY_NAME, "idexists");
    }
    return pointsRepository
        .save(points)
        .flatMap(pointsSearchRepository::save)
        .map(result -> {
            try {
                return ResponseEntity
                    .created(new URI("/api/points/" + result.getId()))
                    .headers(HeaderUtil.createEntityCreationAlert(applicationName, true, ENTITY_NAME, result.getId().toString()))
                    .body(result);
            } catch (URISyntaxException e) {
                throw new RuntimeException(e);
            }
        });
}
```

The code above was created by running `jhipster jdl 21-points.jh --reactive`.

Microservices with JHipster

In this example, I'll show you how to build a reactive microservices architecture with JHipster. As part of this process, you'll generate three applications and run several others.

- Generate a gateway.
- Generate a blog microservice.
- Generate a store microservice.
- Run Consul, Keycloak, Neo4j, and MongoDB.

Introducing Micro Frontends

Before JHipster 7.9.0, if you generated a microservices architecture with a UI, the gateway would be a monolithic UI. This means the gateway would contain all the Angular, React, or Vue files. This creates a tight coupling between the gateway and the microservices it routes to. If you want to change the UI for a microservice, you must also redeploy the gateway. This is a problem because you should be able to deploy your microservices independently.

You can solve this problem with micro frontends. Micro frontends are a way to break up your UI into smaller, independent pieces. JHipster added support for micro-frontends in 7.9.0. Microfrontends provide a way to remotely load and execute code at runtime so your microservice's UI can live in the same artifact without being coupled to the gateway!

In the previous paragraph, I spelled micro frontends three different ways. The current literature is all over the place on this one! I'm going to use "micro frontends" for the remainder of this chapter since that's what Cam Jackson used in his Micro Frontends article on Martin Fowler's blog.

You can see how these components fit in the diagram below.

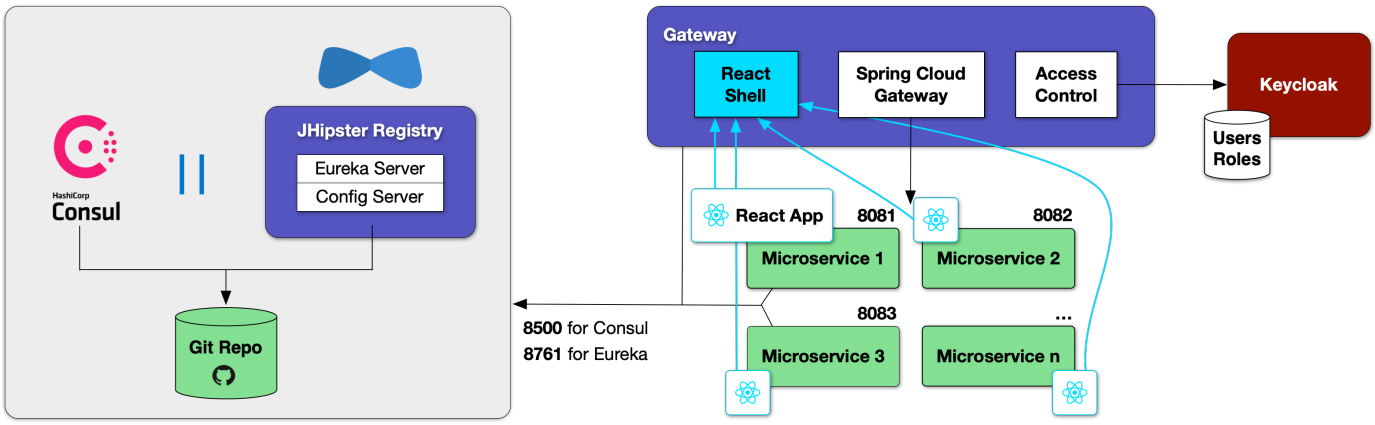


Figure 40. JHipster microservices architecture

This tutorial shows you how to build a microservices architecture with JHipster 7.9.3. You'll generate a gateway (powered by Spring Cloud Gateway), a blog microservice (that talks to Neo4j), and a store microservice (that uses MongoDB). The gateway will contain a React shell app that loads the blog and store micro frontends. You'll use Docker Compose to make sure it all runs locally. I'll also provide some pointers on how to deploy it with Kubernetes.

Generate an API gateway and microservice applications

Open a terminal window, create a directory (e.g., `jhipster-microservices-example`), and create an `apps.jdl` file in it. Copy the JDL below into this file. You can also download this file from GitHub.

Example 1. apps.jdl

```
application {
  config {
    baseName gateway
    reactive true ①
    packageName com.okta.developer.gateway
    applicationType gateway
    authenticationType oauth2 ②
    buildTool gradle
    clientFramework react ③
    prodDatabaseType postgresql
    serviceDiscoveryType consul ④
    testFrameworks [cypress] ⑤
    microfrontends [blog, store] ⑥
  }
}

application {
  config {
    baseName blog
    reactive true
    packageName com.okta.developer.blog
    applicationType microservice ⑦
    authenticationType oauth2 ⑧
    buildTool gradle
    clientFramework react ⑨
    databaseType neo4j ⑩
    devDatabaseType neo4j
    prodDatabaseType neo4j
    enableHibernateCache false
    serverPort 8081 ⑪
    serviceDiscoveryType consul
    testFrameworks [cypress] ⑫
  }
  entities Blog, Post, Tag
}

application {
  config {
```

```
    baseName store
    reactive true
    packageName com.okta.developer.store
    applicationType microservice
    authenticationType oauth2
    buildTool gradle
    clientFramework react
    databaseType mongodb ⑬
    devDatabaseType mongodb
    prodDatabaseType mongodb
    enableHibernateCache false
    serverPort 8082
    serviceDiscoveryType consul
    testFrameworks [cypress]
  }
  entities Product
}

⑭
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
}

⑮
relationship ManyToOne {
  Blog{user(login)} to User
  Post{blog(name)} to Blog
}

relationship ManyToMany {
```

```
  Post{tag(name)} to Tag{post}
}

⑯
paginate Post, Tag with infinite-scroll
paginate Product with pagination

⑰
deployment {
  deploymentType docker-compose
  serviceDiscoveryType consul
  appsFolders [gateway, blog, store]
  dockerRepositoryName "mraible"
}

⑱
deployment {
  deploymentType kubernetes
  appsFolders [gateway, blog, store]
  clusteredDbApps [store]
  kubernetesNamespace demo
  kubernetesUseDynamicStorage true
  kubernetesStorageClassName ""
  serviceDiscoveryType consul
  dockerRepositoryName "mraible"
}
```

① Enable reactive support. You cannot set this to `false` for a gateway. This is because Spring Cloud Gateway is reactive-only. There is an open issue for Spring MVC support.

② The authentication type for the gateway is OAuth 2.0.

③ The client framework used is React.

④ You must specify `consul` as the service discovery type for the gateway and all microservice apps. You can also use `eureka`, but I prefer `consul` because it'll be the default in JHipster 8.

⑤ Including Cypress allows you to test the UI with `npm run e2e`.

⑥ Micro frontends are enabled for the gateway, and entities will be pulled in from the blog and store microservices.

⑦ For the microservice apps, you need to specify an application type of `microservice`.

⑧ The microservice app's authentication type must match the gateway.

⑨ The client framework must be the same for all apps.

⑩ The blog app uses Neo4j as its database. You must use the same databases for dev and prod when using NoSQL options.

⑪ The default server port is 8080. You must specify different ports for each app.

⑫ If you want to test the UI of your micro frontend, you need to include Cypress.

⑬ The store app uses MongoDB for its database.

⑭ Entity definitions live outside your application definitions. You can validate your JDL using JDL-Studio or the JHipster JDL Plugin.

⑮ Relationships between entities can be defined in JDL!

⑯ If you want pagination on your list screens, you can use infinite scrolling or page links.

⑰ Creates Docker Compose files for all apps and a `docker-compose.yml` file that will start them.

⑱ Creates Kubernetes manifests for all apps and scripts to deploy them.

Micro frontend options: Angular, React, and Vue

JHipster has support for the big three JavaScript frameworks: Angular, React, and Vue. All are implemented using TypeScript, and a newly generated app should have around 70% code coverage, both on the backend and frontend.

There is also a Svelte blueprint, but it does not support micro frontends at the time of this writing.

Run JHipster's `jdl` command to import this microservices architecture definition.

```
jhipster jdl apps.jdl --monorepository --workspaces
```

The project generation process will take a minute or two, depending on your internet connection speed and hardware.

The last two arguments are optional, but I expect you to use them for this tutorial. Without the `monorepository` flag, the gateway and microservices would have their own Git repos. The `workspaces` flag enables npm workspaces, which are similar to having an aggregator `pom.xml` that allows you to execute commands across projects. It also makes it so there's only one `node_modules` in the root directory. To learn more, I recommend egghead's Introduction to Monorepos with NPM Workspaces.

If you want to use Angular, append `--client-framework angularX` to the command above to override the JDL value:

```
--client-framework angularX
```

`angularX` is a legacy JDL value from back when JHipster supported AngularJS and Angular 2. We will change it to `angular` in v8.

If you'd rather try out Vue, use the following:

```
--client-framework vue
```

Run your microservices architecture

When the process is complete, cd into the `gateway` directory and start Keycloak and Consul using Docker Compose.

```
cd gateway
docker compose -f src/main/docker/keycloak.yml up -d
docker compose -f src/main/docker/consul.yml up -d
```

Then, run `./gradlew` (or `npm run app:start` if you prefer npm commands). When the startup process completes, open your favorite browser to `http://localhost:8080`, and log in with the credentials displayed on the page.

You'll be redirected back to the gateway, but the **Entities** menu won't have any links because the micro frontends it tries to load are unavailable.

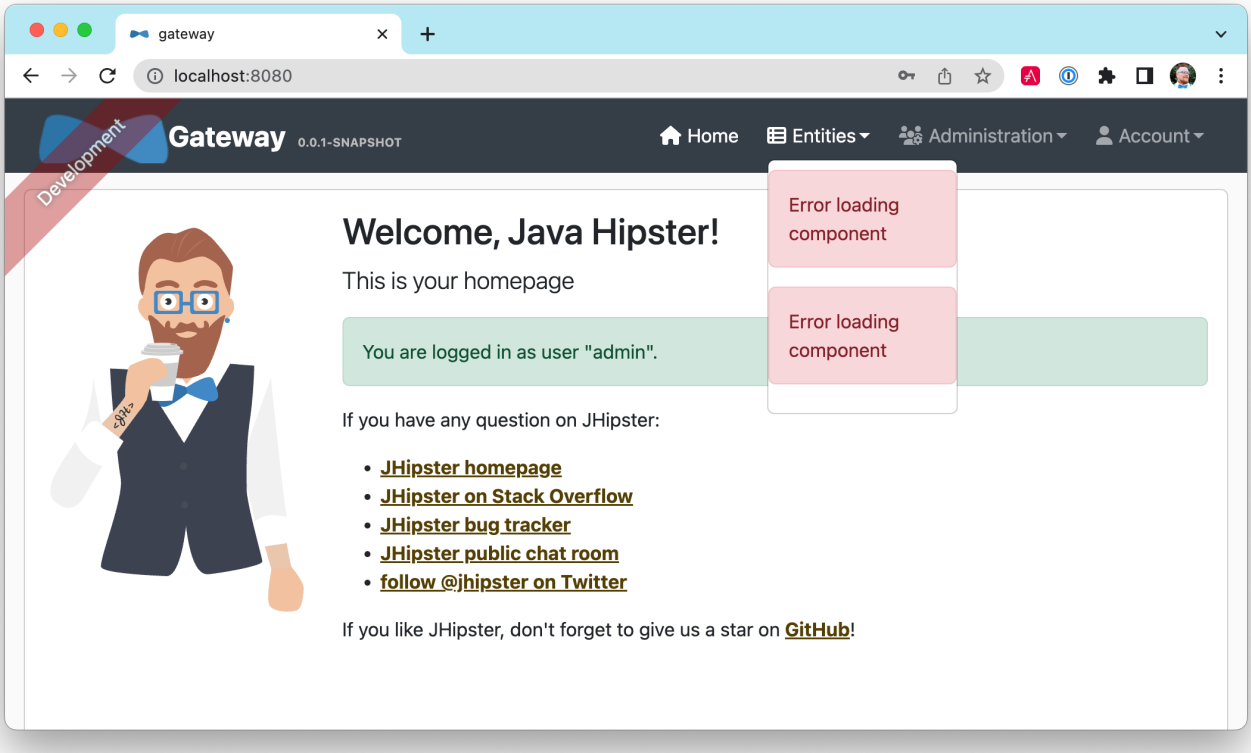


Figure 41. The gateway's entities are unavailable

Start the `blog` by opening a terminal and navigating to its directory. Then, start its database with

Docker and the app with Gradle.

```
npm run docker:db:up
./gradlew
```

Open a new terminal and do the same for the `store` microservice.

You can verify everything is started using Consul at `http://localhost:8500`.

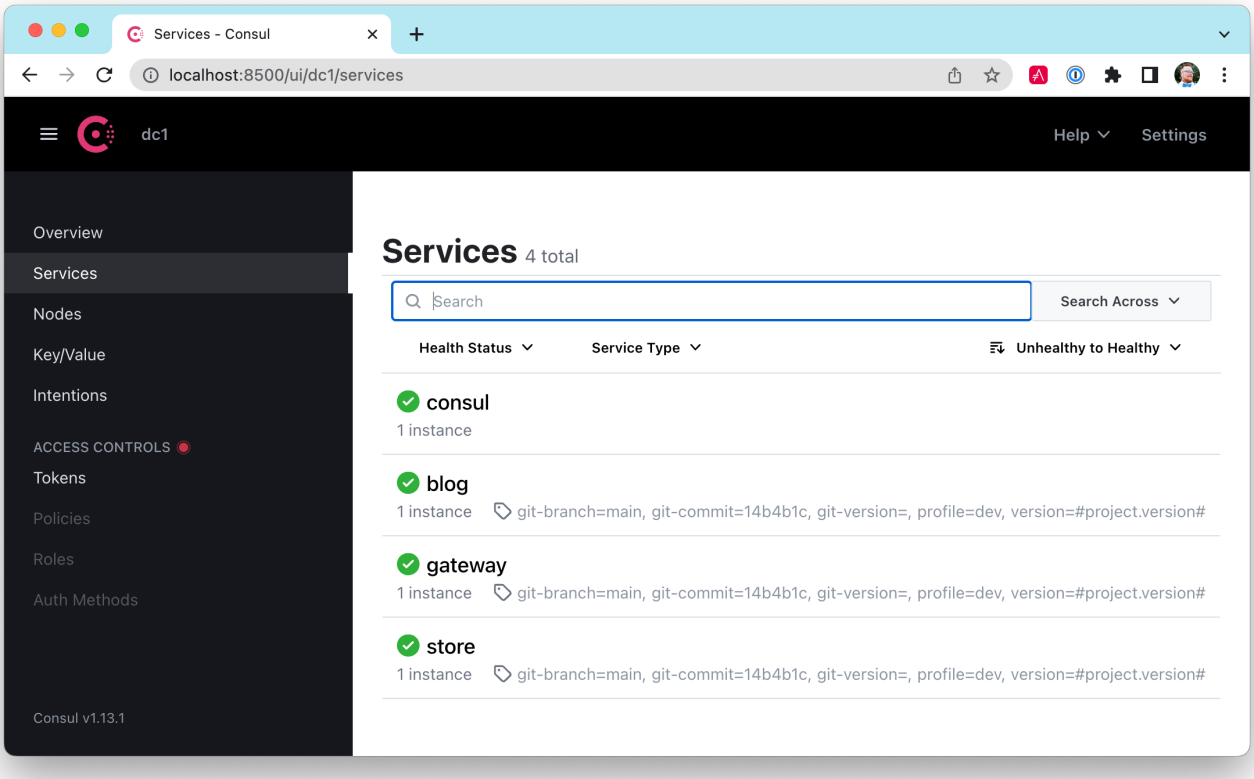


Figure 42. Consul services

Refresh the gateway app; you should see menu items to navigate to the microservices now.

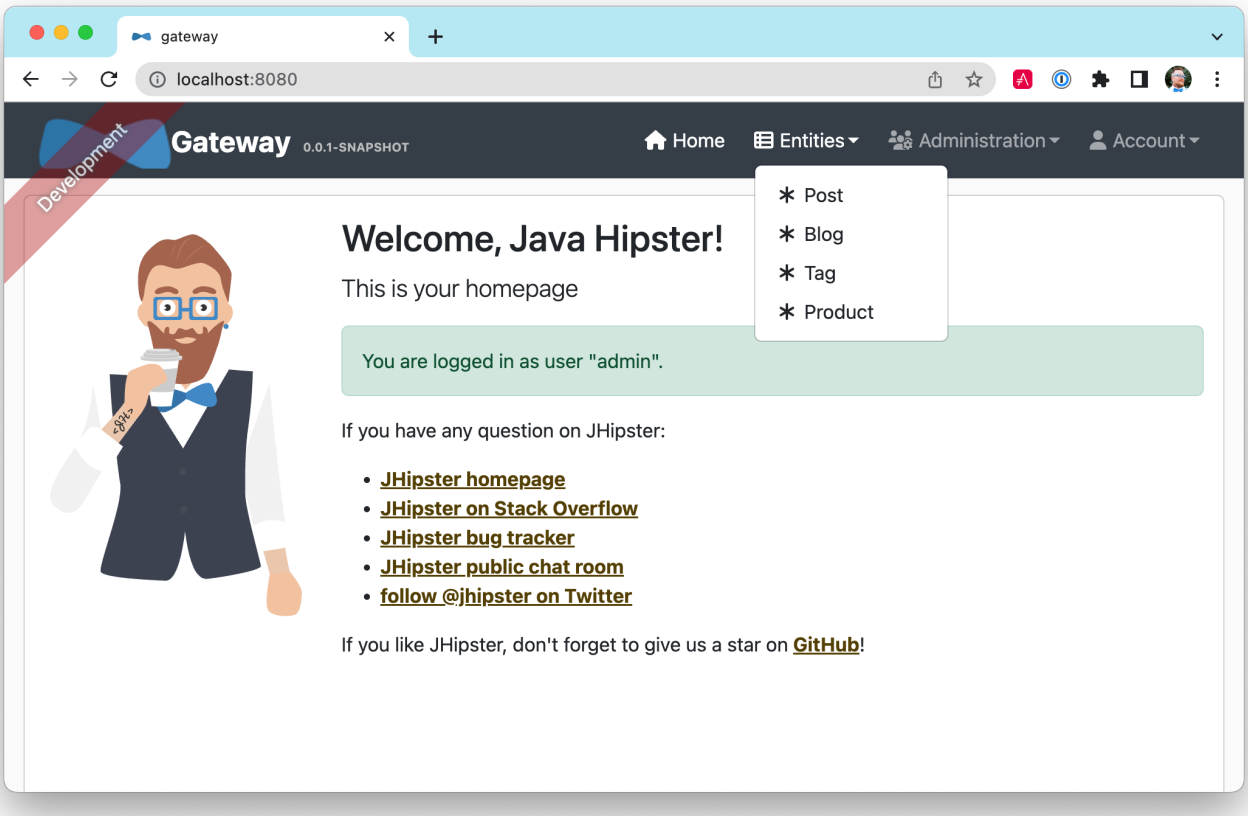


Figure 43. Gateway entities available

Zero turnaround development that sparks joy

At this point, I've only shown you how to run the Spring Boot backends with their packaged React micro frontends. What if you want to work on the UI and have zero turnaround that sparks joy?

In the gateway app, run `npm start`. This command will run the UI on a web server, open a browser window to `http://localhost:9000`, and use Browsersync to keep your browser in sync with your code.

Modify the code in `gateway/src/main/webapp/app/modules/home/home.tsx` to make a quick change. For example, add the following HTML below the `<h2>`.

```
<h3 className="text-primary">
  Hi, I'm a quick edit!
</h3>
```

You'll see this change immediately appear within your browser.

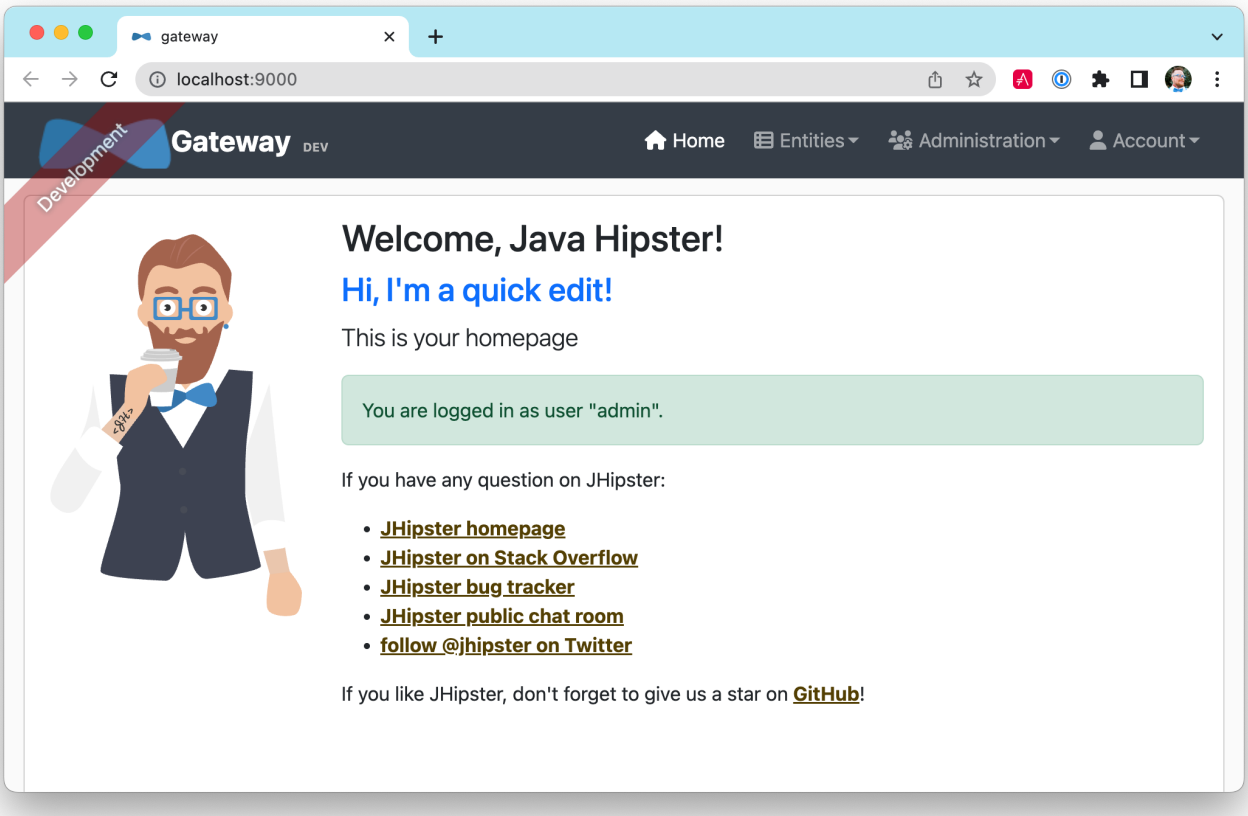


Figure 44. Gateway quick edit

Remove it, and it'll disappear right away too.

Now, open another terminal and navigate into the `store` directory. Run `npm start`, and you'll have a similar zero-turnaround experience when modifying files in the `store` app. The app will start a webserver on `http://localhost:9002`, and there will only be one menu item for product. Modify files in the `store/src/main/webapp/app/entities/store/product` directory, and you'll see the changes in your browser immediately. For example, change the wrapper `<div>` in `product.tsx` to have a background color:

```
<div className="bg-info">
```

The UI will change before you can `Cmd+Tab` back to your browser.

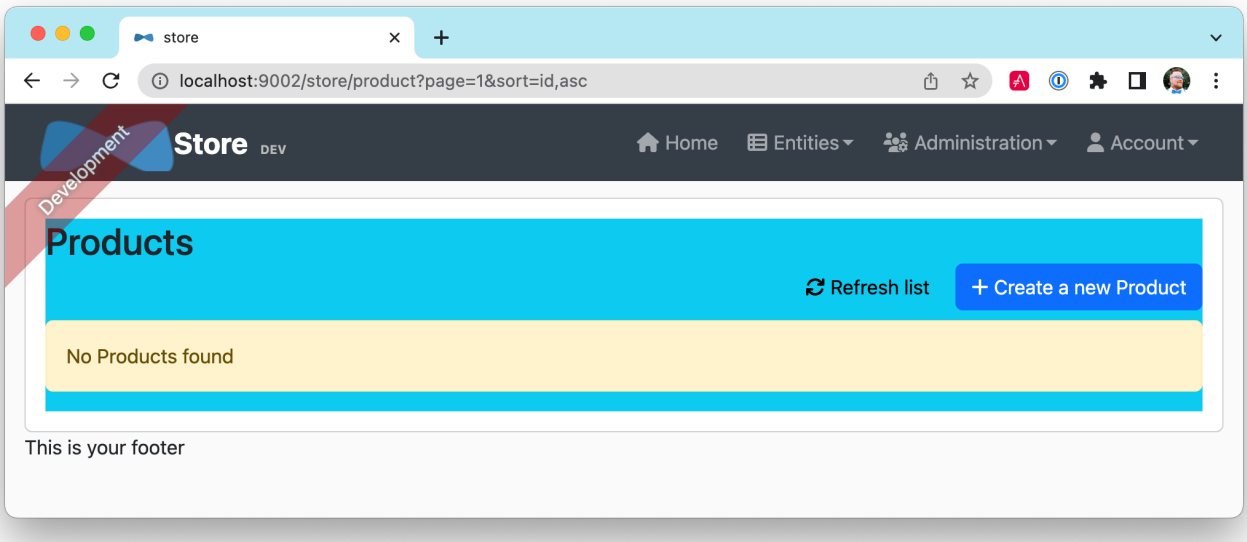


Figure 45. Store edit

The backend has quick turnaround abilities, too, thanks to Spring Boot devtools. If you modify a backend class, recompiling it will cause Spring Boot to reload your component lickety-split. It's pretty slick!

A look under the hood of micro frontends

When you're learning concepts like micro frontends, it's often helpful to look at the code that makes things work.

The gateway's `webpack.microfrontend.js` handles setting up the `@blog` and `@store` remotes and specifying the shared dependencies and components between apps.

Listing 65. `gateway/webpack/webpack.microfrontend.js`

```
const ModuleFederationPlugin = require('webpack/lib/container/ModuleFederationPlugin');

const packageJson = require('../package.json');
const appVersion = packageJson.version;

module.exports = ({ serve }) => {
  return {
    optimization: {
      moduleIds: 'named',
      chunkIds: 'named',
      runtimeChunk: false,
    },
    plugins: [
      new ModuleFederationPlugin({
```

```
        shareScope: 'default',
        remotes: {
          '@blog': `blog@/services/blog/remoteEntry.js`,
          '@store': `store@/services/store/remoteEntry.js`,
        },
        shared: {
          ...Object.fromEntries(
            Object.entries(packageJson.dependencies).map(([module, version]) => [
              module,
              { requiredVersion: version, singleton: true, shareScope: 'default' },
            ])
          ),
          'app/config/constants': {
            singleton: true,
            import: 'app/config/constants',
            requiredVersion: appVersion,
          },
          'app/config/store': {
            singleton: true,
            import: 'app/config/store',
            requiredVersion: appVersion,
          },
          'app/shared/error/error-boundary-routes': {
            singleton: true,
            import: 'app/shared/error/error-boundary-routes',
            requiredVersion: appVersion,
          },
          'app/shared/layout/menus/menu-components': {
            singleton: true,
            import: 'app/shared/layout/menus/menu-components',
            requiredVersion: appVersion,
          },
          'app/shared/layout/menus/menu-item': {
            singleton: true,
            import: 'app/shared/layout/menus/menu-item',
            requiredVersion: appVersion,
          },
          'app/shared/reducers': {
            singleton: true,
            import: 'app/shared/reducers',
            requiredVersion: appVersion,
          },
          'app/shared/reducers/locale': {
            singleton: true,
            import: 'app/shared/reducers/locale',
            requiredVersion: appVersion,
          },
```

```
          'app/shared/reducers/reducer.utils': {
            singleton: true,
            import: 'app/shared/reducers/reducer.utils',
            requiredVersion: appVersion,
          },
          'app/shared/util/date-utils': {
            singleton: true,
            import: 'app/shared/util/date-utils',
            requiredVersion: appVersion,
          },
          'app/shared/util/entity-utils': {
            singleton: true,
            import: 'app/shared/util/entity-utils',
            requiredVersion: appVersion,
          },
        },
      }),
    ],
    output: {
      publicPath: 'auto',
    },
  };
};
```

The blog's `webpack.microfrontend.js` looks similar, except that it exposes its `remoteEntry.js`, menu items, and routes.

Listing 66. `blog/webpack/webpack.microfrontend.js`

```
const ModuleFederationPlugin = require('webpack/lib/container/ModuleFederationPlugin');
const { DefinePlugin } = require('webpack');

const packageJson = require('../package.json');
const appVersion = packageJson.version;

module.exports = ({ serve }) => {
  return {
    optimization: {
      moduleIds: 'named',
      chunkIds: 'named',
      runtimeChunk: false,
    },
    plugins: [
      new ModuleFederationPlugin({
        name: 'blog',
        filename: 'remoteEntry.js',
```

```
        shareScope: 'default',
        exposes: {
          './entities-menu': './src/main/webapp/app/entities/menu',
          './entities-routes': './src/main/webapp/app/entities/routes',
        },
        shared: {
          ...Object.fromEntries(
            Object.entries(packageJson.dependencies).map(([module, version]) => [
              module,
              { requiredVersion: version, singleton: true, shareScope: 'default' },
            ])
          ),
          'app/config/constants': {
            singleton: true,
            import: 'app/config/constants',
            requiredVersion: appVersion,
          },
          'app/config/store': {
            singleton: true,
            import: 'app/config/store',
            requiredVersion: appVersion,
          },
          'app/shared/error/error-boundary-routes': {
            singleton: true,
            import: 'app/shared/error/error-boundary-routes',
            requiredVersion: appVersion,
          },
          'app/shared/layout/menus/menu-components': {
            singleton: true,
            import: 'app/shared/layout/menus/menu-components',
            requiredVersion: appVersion,
          },
          'app/shared/layout/menus/menu-item': {
            singleton: true,
            import: 'app/shared/layout/menus/menu-item',
            requiredVersion: appVersion,
          },
          'app/shared/reducers': {
            singleton: true,
            import: 'app/shared/reducers',
            requiredVersion: appVersion,
          },
          'app/shared/reducers/locale': {
            singleton: true,
            import: 'app/shared/reducers/locale',
            requiredVersion: appVersion,
          },
```

```
          'app/shared/reducers/reducer.utils': {
            singleton: true,
            import: 'app/shared/reducers/reducer.utils',
            requiredVersion: appVersion,
          },
          'app/shared/util/date-utils': {
            singleton: true,
            import: 'app/shared/util/date-utils',
            requiredVersion: appVersion,
          },
          'app/shared/util/entity-utils': {
            singleton: true,
            import: 'app/shared/util/entity-utils',
            requiredVersion: appVersion,
          },
        },
      }),
      new DefinePlugin({
        BLOG_I18N_RESOURCES_PREFIX: JSON.stringify(''),
      }),
    ],
    output: {
      publicPath: 'auto',
    },
  };
};
```

Build and run with Docker

To build Docker images for each application, run the following command from the root directory.

```
npm run java:docker
```

The command is slightly different if you're using a Mac with Apple Silicon.

```
npm run java:docker:arm64
```

You can see all npm scripts with `npm run`.

Then, navigate to the `docker-compose` directory, stop the existing containers, and start all the containers.

```
cd docker-compose
docker stop $(docker ps -a -q);
```

```
docker compose up
```

This command will start and run all the apps, their databases, Consul, and Keycloak. To make Keycloak work, you must add the following line to your hosts file (`/etc/hosts` on Mac/Linux, `c:\Windows\System32\Drivers\etc\hosts` on Windows).

```
127.0.0.1  keycloak
```

This is because you will access your application with a browser on your machine (where the name is localhost, or `127.0.0.1`), but inside Docker, it will run in its own container, where the name is `keycloak`.

If you want to prove everything works, ensure everything is started at `http://localhost:8500`, then run `npm run e2e -ws` from the root project directory. This command will run the Cypress tests that JHipster generates in your browser.

Switch identity providers

JHipster ships with Keycloak when you choose OAuth 2.0 / OIDC as the authentication type. However, you can easily change it to another identity provider, like Auth0!

First, you'll need to register a regular web application. Log in to your Auth0 account (or sign up if you don't have an account). You should have a unique domain like `dev-xxx.us.auth0.com`.

Select **Create Application** in the Applications section. Use a name like `Micro Frontends`, select **Regular Web Applications**, and click **Create**.

Switch to the **Settings** tab and configure your application settings:

- Allowed Callback URLs: `http://localhost:8080/login/oauth2/code/oidc`
- Allowed Logout URLs: `http://localhost:8080/`

Scroll to the bottom and click **Save Changes**.

In the roles section, create new roles named `ROLE_ADMIN` and `ROLE_USER`.

Create a new user account in the users section. Click the **Role** tab to assign the roles you just created to the new account.

Make sure your new user's email is verified before logging in!

Next, head to **Actions** > **Flows** and select **Login**. Create a new action named `Add Roles` and use the default trigger and runtime. Change the `onExecutePostLogin` handler to:

```
exports.onExecutePostLogin = async (event, api) => {
```

```
  const namespace = 'https://www.jhipster.tech';
  if (event.authorization) {
    api.idToken.setCustomClaim('preferred_username', event.user.email);
    api.idToken.setCustomClaim(`${namespace}/roles`, event.authorization.roles);
    api.accessToken.setCustomClaim(`${namespace}/roles`, event.authorization.roles);
  }
}
```

This code adds the user's roles to a custom claim (prefixed with `https://www.jhipster.tech/roles`). This claim is mapped to Spring Security authorities in `SecurityUtils.java` in the gateway app.

Select **Deploy** and drag the `Add Roles` action to your Login flow.

Edit `docker-compose/central-server-config/application.yml` and append the following YAML block to add your Auth0 settings.

```
jhipster:
  security:
    oauth2:
      audience: https://<your-auth0-domain>/api/v2/

spring:
  security:
    oauth2:
      client:
        provider:
          oidc:
            issuer-uri: https://<your-auth0-domain>/
        registration:
          oidc:
            client-id: <your-client-id>
            client-secret: <your-client-secret>
```

Want to have all these steps automated for you? Vote for issue #351 in the Auth0 CLI project.

Stop all your Docker containers with `Ctrl+C` and start them again.

```
docker compose up
```

Now, Spring Security will be configured to use Auth0, and Consul will distribute these settings to all your microservices. When everything is started, navigate to `http://localhost:8080` and click **sign in**. You will be prompted for your Auth0 credentials.

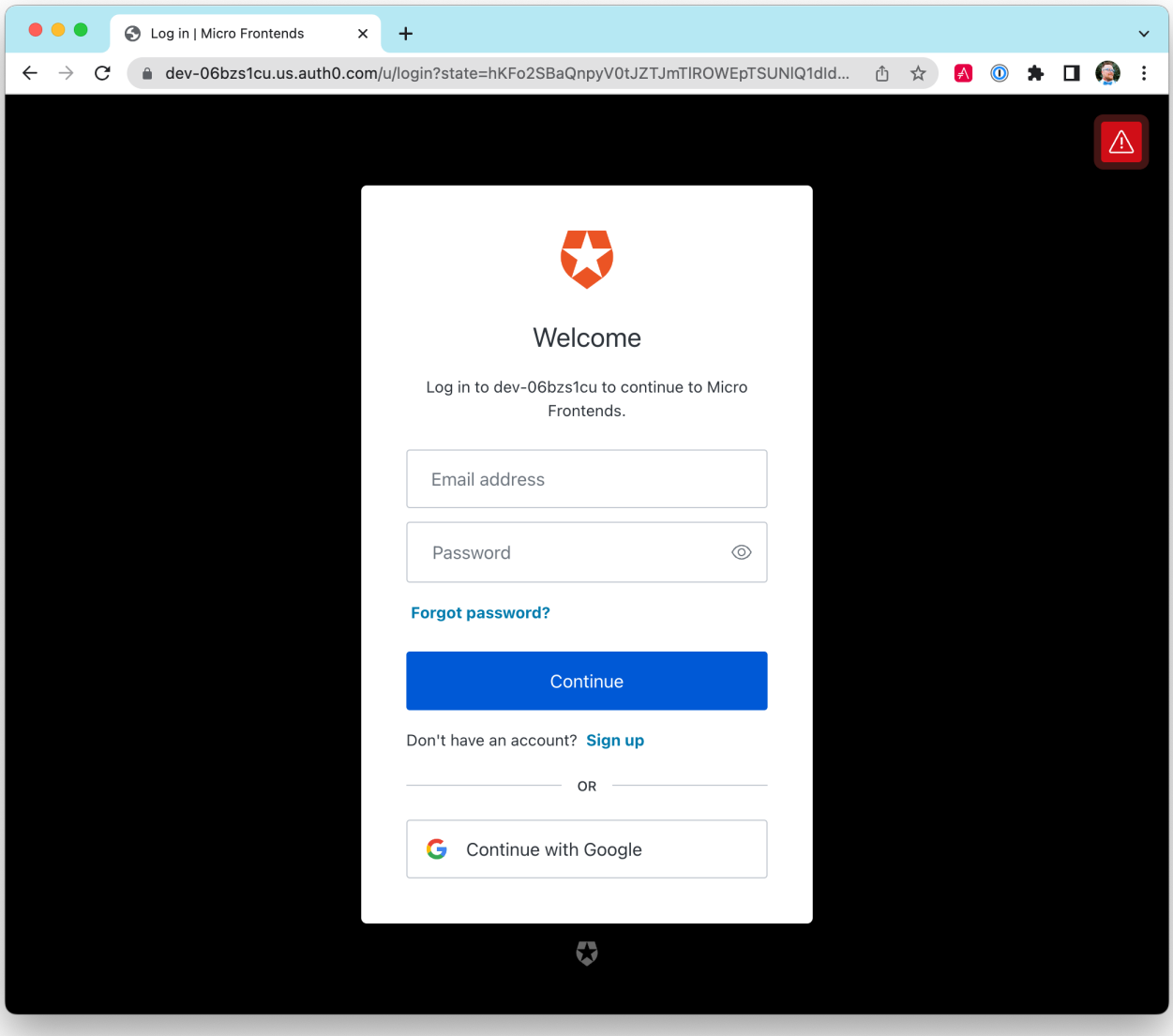


Figure 46. Auth0 login

After entering your credentials, you'll be redirected back to the gateway, and your username will be displayed.

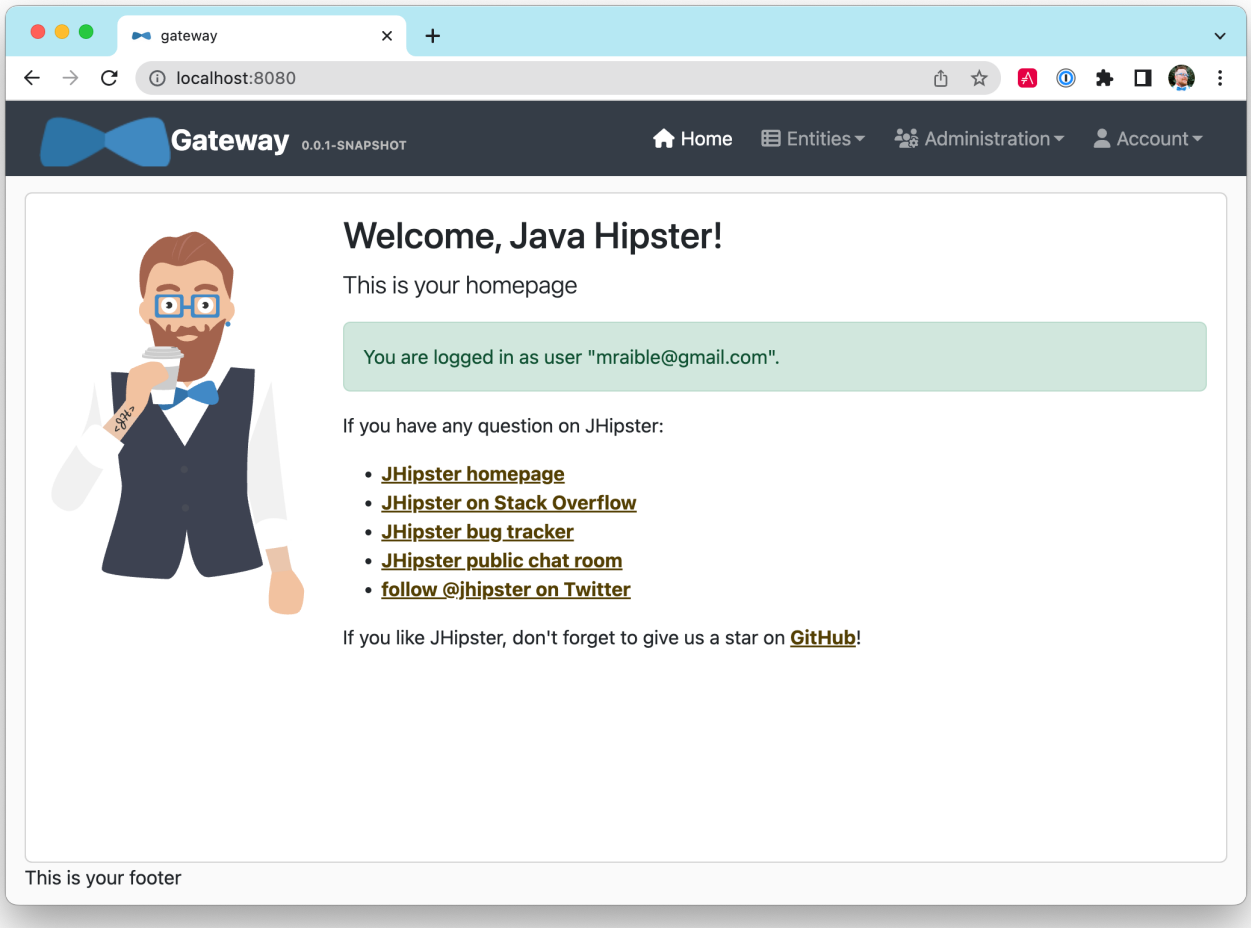


Figure 47. Auth0 login success

You should be able to add, edit, and delete blogs, posts, tags, and products, proving that your microservices and micro frontends can talk to each other.

If you'd like to use Okta for your identity provider, see JHipster's documentation.

You can configure JHipster quickly with the Okta CLI:

```
okta apps create jhipster
```

Deploy with Kubernetes

The JDL you used to generate this microservices stack has a section at the bottom for deploying to Kubernetes.

```
deployment {
  deploymentType kubernetes
```

```
  appsFolders [gateway, blog, store]
  clusteredDbApps [store]
  kubernetesNamespace demo
  kubernetesUseDynamicStorage true
  kubernetesStorageClassName ""
  serviceDiscoveryType consul
  dockerRepositoryName "mraible"
}
```

The `jhipster jdl` command generates a `kubernetes` directory with this information and configures all your apps, databases, and Consul to be Kubernetes-ready. If you have a Kubernetes cluster created, you can deploy to its `demo` namespace using the following command.

```
./kubectl-apply.sh -f
```

It also generates files for Kustomize and Skaffold if you'd prefer to use those tools. See the `kubernetes/K8S-README.md` file for more information.

I won't go into the nitty-gritty details of deploying a JHipster microservices stack to cloud providers with K8s, mainly because it's covered in other guides. The first one below shows how to run Minikube locally, encrypt your secrets, and deploy to Google Cloud.

- Deploy JHipster Microservices to GCP with Kubernetes
- Create Kubernetes Microservices on Azure with Cosmos DB
- Run Microservices on DigitalOcean with Kubernetes
- How to Deploy JHipster Microservices on Amazon EKS Using Terraform and Kubernetes
- CI/CD Java Microservices with CircleCI and Spinnaker

Source code

You can find the source code for this microservices example at @oktadev/auth0-micro-frontends-jhipster-example.

Summary

I hope you enjoyed this overview of how to use micro frontends within a Java microservices architecture. I like how micro frontends allow each microservice application to be self-contained and deployable; independent of the other microservices. It's also neat how JHipster generates Docker and Kubernetes configurations for you. Cloud-native FTW!

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.

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 six simple policies, paraphrased here:

1. The development team votes on policies.
2. Use technologies with their default configuration and best practices as much as possible.
3. Only add options when there is sufficient added value in the generated code.
4. Use strict versions for third-party libraries.
5. Provide similar user/developer experience across different options as much as possible.
6. Developer experience can take precedence over above policies.

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, Spring Boot, and microservices with JHipster, go forth and develop awesome applications!

Additional reading

If you want to learn more, here are some suggestions.

I wrote *The Angular Mini-Book* (InfoQ, February 2022) as a bare-bones guide to using Angular. It's a good place to start if you're new to Angular. It also includes extensive Spring Boot coverage, security best practices, and cloud deployment options.

Deepu K Sasidharan and Sendil Kumar N, two prolific committers on the JHipster team, wrote *Full Stack Development with JHipster: Second Edition* (Packt Publishing, January 2020). This book covers JHipster 6.

Learn how to use Spring Boot to be productive and build mission-critical applications with *Spring Boot: Up and Running* (O'Reilly Media, February 2021) by Mark Heckler.

Josh Long's Reactive Spring (self-published, February 2022) introduces reactive programming and its implementation in the Spring ecosystem. If you want to learn more about WebFlux, this is a great place to start.

One of the most comprehensive books I've read on Angular is *ng-book: The Complete Book on Angular* by Nathan Murray, Felipe Coury, Ari Lerner, and Carlos Taborda) (Fullstack.io, continuously updated).

About the author

Matt Raible is a Java Hipster. He grew up in the backwoods of Montana with no electricity or running water. On school days, he walked a mile and a half to the bus stop. 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 is passionate about 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 LinkedIn (linkedin.com/in/mraible) and Twitter (@mraible). Matt drives a 1966 21-Window Bus and a 1990 Vanagon Syncro.