THE JHIPSTER MINI-BOOK

Matt Raible

InfoQueue
ENTERPRISE SOFTWARE
DEVELOPMENT SERIES

THE JHIPSTER MINI-BOOK

Matt Raible

The JHipster Mini-Book

Published by C4Media, publisher of InfoQ.com.

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

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

Table of Contents

Dedication

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

Thanks, Mom and Dad - you rock!

Acknowledgements

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

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

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

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

Finally, kudos to Julien Dubois and Deepu 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

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

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

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

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

What is in an InfoQ mini-book?

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

Who this book is for

This book is aimed specifically at web developers who want a rapid introduction to Angular, Bootstrap, and Spring Boot by learning JHipster.

What you need for this book

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

Conventions

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

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

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

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

src/app/search/search.component.html

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

src/main/java/demo/DemoApplication.java

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

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

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

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

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

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

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

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

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

Tips are shown using callouts like this.

Warnings are shown using callouts like this.

Sidebar

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

Finally, this text shows what a quote looks like:

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

Reader feedback

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

To send us feedback, e-mail us at feedback@infoq.com, send a tweet to @jhipster_book, or post a question on Stack Overflow using the "jhipster" tag.

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

Introduction

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

JHipster was started by Julien Dubois in October 2013 (Julien’s first commit was on October 21, 2013). The first public release (version 0.3.1) was launched December 7, 2013. Since then, the project has had over 146 releases! It is an open-source, Apache 2.0-licensed project on GitHub. It has a core team of 19 developers and over 350 contributors. You can find its homepage at http://www.jhipster.tech. Its GitHub project shows it’s mostly written in JavaScript (40%), Java (29%), and HTML (14%). TypeScript is trailing in the fourth position with 13%.

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

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

- the scaffolding tool (yo),
- the build tool (Gulp, npm/yarn, webpack, etc.), and
- the package manager (Bower, npm/yarn, etc.).

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

PART ONE

Building an app with JHipster

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

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

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

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

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

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

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

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

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

I thought writing a "21-Point Health" application would work because tracking your health is always important. Wearables that can track your health stats might be able to use the APIs or hooks I create to

record points for a day. Imagine hooking into Dailymile (where I track my exercise) or Untappd (where I sometimes list the beers I drink). Or even displaying other activities for a day (e.g. showing your blood-pressure score that you keep on iOS Health).

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

Creating the application

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

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

If you need to install Java, Maven, or Git, please see JHipster's local installation documentation.

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

```
yo jhipster
```

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

Figure 1. Generating the application

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

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

yo-rc.json

```
{
  "generator-jhipster": {
    "promptValues": {
      "packageName": "org.jhipster.health",
      "nativeLanguage": "en"
    },
    "jhipsterVersion": "4.5.2",
    "baseName": "TwentyOnePoints",
    "packageName": "org.jhipster.health",
    "packageFolder": "org/jhipster/health",
    "serverPort": "8080",
    "authenticationType": "jwt",
    "hibernateCache": "ehcache",
    "clusteredHttpSession": false,
    "websocket": false,
    "databaseType": "sql",
    "devDatabaseType": "h2Disk",
    "prodDatabaseType": "postgresql",
    "searchEngine": "elasticsearch",
    "messageBroker": false,
    "serviceDiscoveryType": false,
    "buildTool": "gradle",
    "enableSocialSignIn": false,
    "jwtSecretKey": "51b88e52708cfa62c9bc4a7e6c50eff6143ad55c",
    "clientFramework": "angular2",
    "useSass": true,
    "clientPackageManager": "yarn",
    "applicationType": "monolith",
    "testFrameworks": [
      "gatling",
      "protractor"
    ],
    "jhiPrefix": "jhi",
    "enableTranslation": true,
    "nativeLanguage": "en",
    "languages": [
      "en",
      "fr"
    ]
  }
}
```

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

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

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

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

```
BUILD SUCCESSFUL

Total time: 1 mins 49.367 secs
```

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

```
$ yarn e2e
yarn e2e v0.24.4
$ protractor src/test/javascript/protractor.conf.js
(node:82790) [DEP0022] DeprecationWarning: os.tmpDir() is deprecated. Use os.tmpdir()
instead.
[21:18:47] W/configParser - pattern ./e2e/entities/*.spec.ts did not match any files.
[21:18:47] I/launcher - Running 1 instances of WebDriver
[21:18:47] I/direct - Using ChromeDriver directly...
Started
..........


10 specs, 0 failures
Finished in 17.828 seconds

[21:19:08] I/launcher - 0 instance(s) of WebDriver still running
[21:19:08] I/launcher - chrome #01 passed
    Done in 22.14s.
```

To prove the `prod` profile worked and I could talk to PostgreSQL, I installed Postgres.app and tried creating a local PostgreSQL database with settings from `src/main/resources/config/application-prod.yml`.

```
"/Applications/Postgres.app/Contents/Versions/9.6/bin/psql" -p5432 -d "postgres"
  104% [mraible:~] $ "/Applications/Postgres.app/Contents/Versions/9.6/bin/psql" -p5432
-d "postgres"
psql (9.6.3)
Type "help" for help.
postgres=# create user TwentyOnePoints with password '21points';
CREATE ROLE
postgres=# create database TwentyOnePoints;
CREATE DATABASE
postgres=# grant all privileges on database TwentyOnePoints to TwentyOnePoints;
GRANT
postgres=#
```

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

```
$ ./gradlew -Pprod
...
2017-06-05 21:36:47.801 ERROR 85865 --- [           main] o.s.boot.SpringApplication
: Application startup failed

org.springframework.beans.factory.BeanCreationException: Error creating bean with
name 'liquibase' defined in class path resource
[org/jhipster/health/config/DatabaseConfiguration.class]: Invocation of init method
failed; nested exception is liquibase.exception.DatabaseException: 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!

```
$ ./gradlew -Pprod
...
----------------------------------------------------------
        Application 'TwentyOnePoints' is running! Access URLs:
        Local:          http://localhost:8080
        External:       http://192.168.1.27:8080
        Profile(s):     [prod]
----------------------------------------------------------
```

Adding source control

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

The following commands show how I initialized Git, committed the project, added a reference to the remote Bitbucket repository, then pushed everything. I created the repository on Bitbucket before executing these commands.

```
$ git init
Initialized empty Git repository in /Users/mraible/dev/21-points/.git/

$ git remote add origin git@bitbucket.org:mraible/21-points.git

$ git add .

$ git commit -m "Initial checkin of 21-points application"
[master (root-commit) 447c712] Initial checkin of 21-points application
 353 files changed, 27405 insertions(+)
 ...

$ git push origin master
Counting objects: 471, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (448/448), done.
Writing objects: 100% (471/471), 368.52 KiB | 0 bytes/s, done.
Total 471 (delta 58), reused 0 (delta 0)
To bitbucket.org:mraible/21-points.git
 * [new branch]      master -> master
```

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

Building the UI and business logic

I wanted 21-Points Health to be a bit more hip than a stock JHipster application. Bootstrap was all the rage a couple of 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.x. I also discovered the project it depends on is still in development for Bootstrap v4. Since Bootstrap 4 is still an alpha release at the time of this writing, I

opted to simply use Bootstrap and its default theme.

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

Generating entities

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

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

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

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

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

If you want to see how how easy it is to use JDL-Studio, please see this YouTube demo.

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

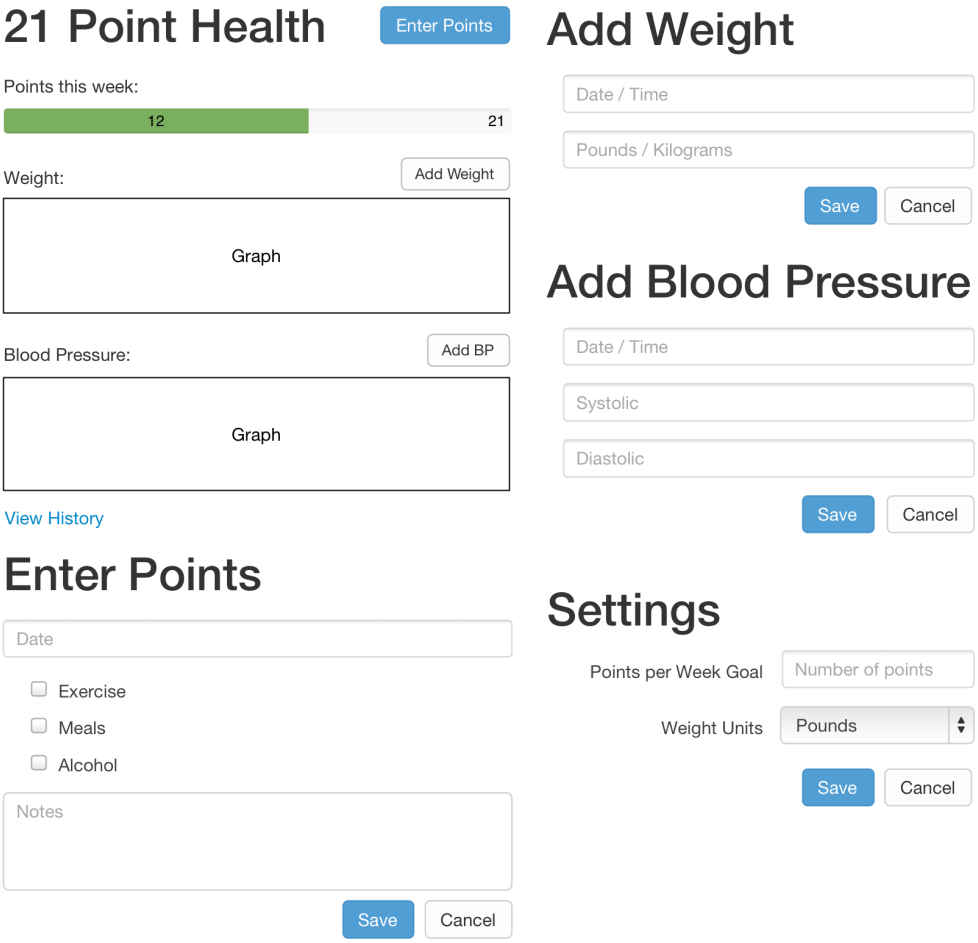

Figure 2. UI mockup

After figuring out how I wanted the UI to look, I started to think about the data model. I quickly decided I didn't need to track high-level goals (e.g. lose ten pounds in Q3 2017). 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.

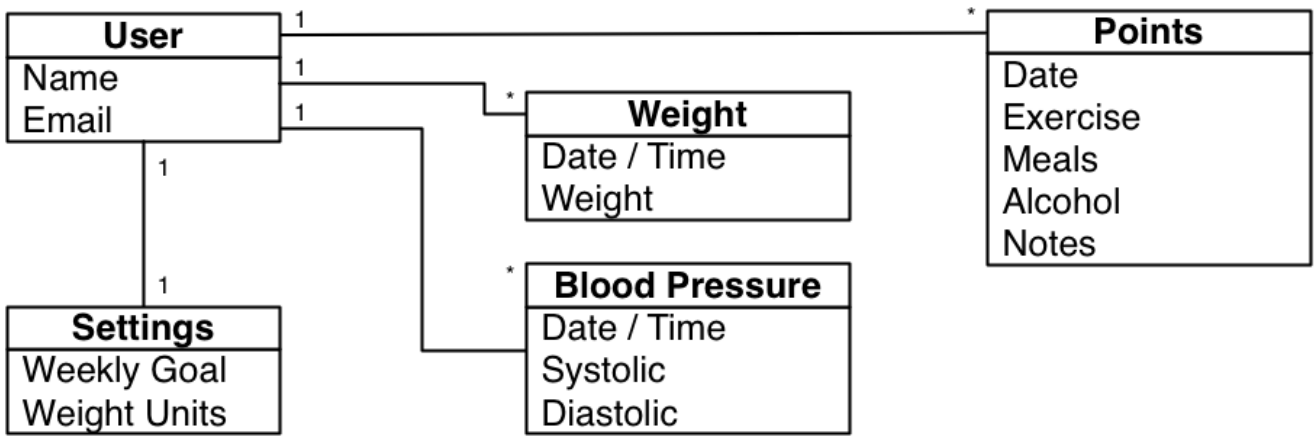

Figure 3. 21-Points Health entity diagram

I ran `yo 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 a Data Transfer Object (DTO)? No, use the entity directly
? Do you want to use separate service class for your business logic? No, the REST
controller should
use the repository directly
? Do you want pagination on your entity? Yes, with pagination links

Everything is configured, generating the entity...

   create .jhipster/Points.json
   create
src/main/resources/config/liquibase/changelog/20170725030037_added_entity_Points.xml
   create
src/main/resources/config/liquibase/changelog/20170725030037_added_entity_constraints_Poi
nts.xml
   create src/main/java/org/jhipster/health/domain/Points.java
   create src/main/java/org/jhipster/health/repository/PointsRepository.java
   create src/main/java/org/jhipster/health/web/rest/PointsResource.java
   create src/main/java/org/jhipster/health/repository/search/PointsSearchRepository.java
   create src/test/java/org/jhipster/health/web/rest/PointsResourceIntTest.java
   create src/test/gatling/user-files/simulations/PointsGatlingTest.scala
 conflict src/main/resources/config/liquibase/master.xml
? Overwrite src/main/resources/config/liquibase/master.xml? overwrite this and all others
    force src/main/resources/config/liquibase/master.xml
    force src/main/java/org/jhipster/health/config/CacheConfiguration.java
   create src/main/webapp/app/entities/points/points.component.html
   create src/main/webapp/app/entities/points/points-detail.component.html
   create src/main/webapp/app/entities/points/points-dialog.component.html
   create src/main/webapp/app/entities/points/points-delete-dialog.component.html
   create src/main/webapp/app/entities/points/index.ts
   create src/main/webapp/app/entities/points/points.module.ts
   create src/main/webapp/app/entities/points/points.route.ts
   create src/main/webapp/app/entities/points/points.model.ts
   create src/main/webapp/app/entities/points/points.component.ts
   create src/main/webapp/app/entities/points/points-dialog.component.ts
```

```
   create src/main/webapp/app/entities/points/points-delete-dialog.component.ts
   create src/main/webapp/app/entities/points/points-detail.component.ts
   create src/main/webapp/app/entities/points/points.service.ts
   create src/main/webapp/app/entities/points/points-popup.service.ts
   create src/test/javascript/spec/app/entities/points/points-detail.component.spec.ts
   create src/test/javascript/e2e/entities/points.spec.ts
    force src/main/webapp/app/entities/entity.module.ts
    force src/main/webapp/app/layouts/navbar/navbar.component.html
   create src/main/webapp/i18n/en/points.json
    force src/main/webapp/i18n/en/global.json
   create src/main/webapp/i18n/fr/points.json
    force src/main/webapp/i18n/fr/global.json

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

I had similar answers for the `Weight` and `BloodPressure` entities. 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
weekly_goal (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? weight_units
? 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)? kg,lb
? Do you want to add validation rules to your field? Yes
? Which validation rules do you want to add? Required

================= Preferences =================
Fields
weekly_goal (Integer) required min='10' max='21'
weight_units (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 `yo 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

Total time: 1 mins 0.004 secs
```

I checked in six changed files and 113 new files generated by the JHipster before continuing to implement my UI mockups.

Application improvements

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

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

Fixed issues with variable names

For the `Preferences` entity, I specified `weekly_goals` and `weight_unit` as field names. I was thinking in terms of names for database columns when I chose these names. I later learned that these names were used throughout my code. I left the column names intact and manually renamed everything in Java, JavaScript, JSON, and HTML to `weeklyGoals` and `weightUnit`.

Improved HTML layout and I18N messages

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

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

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

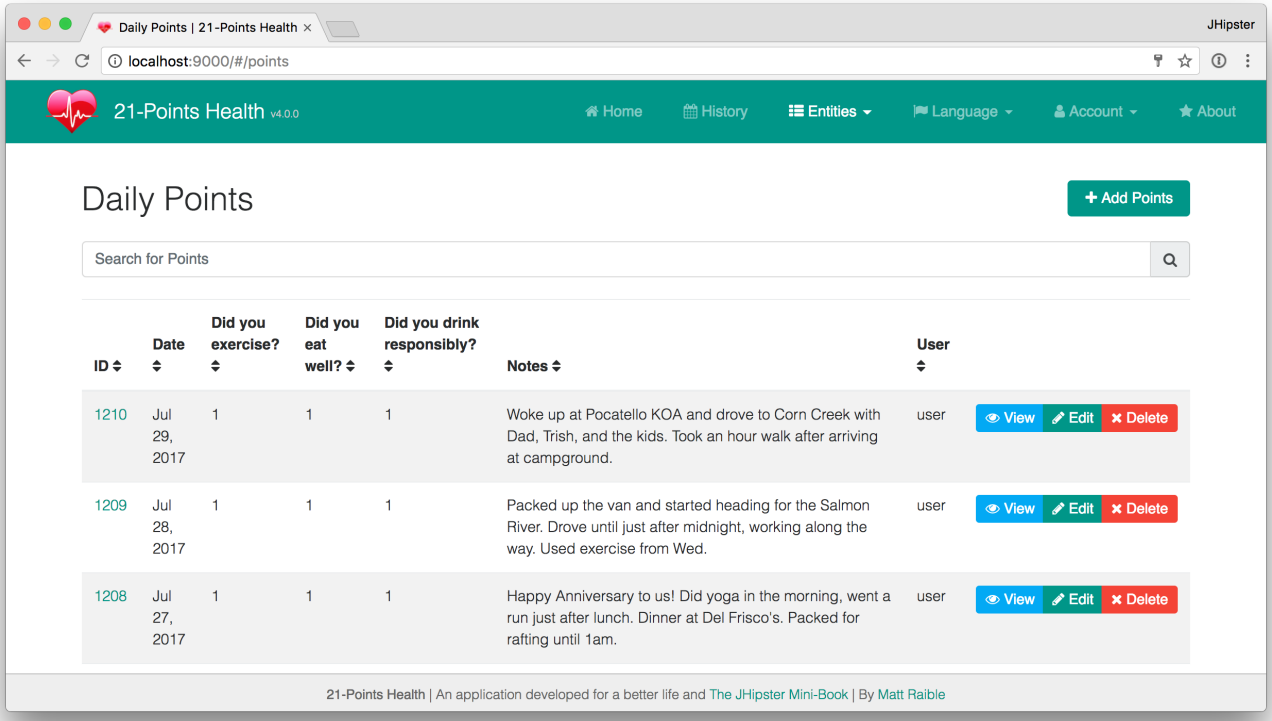

Figure 4. Default Daily Points list

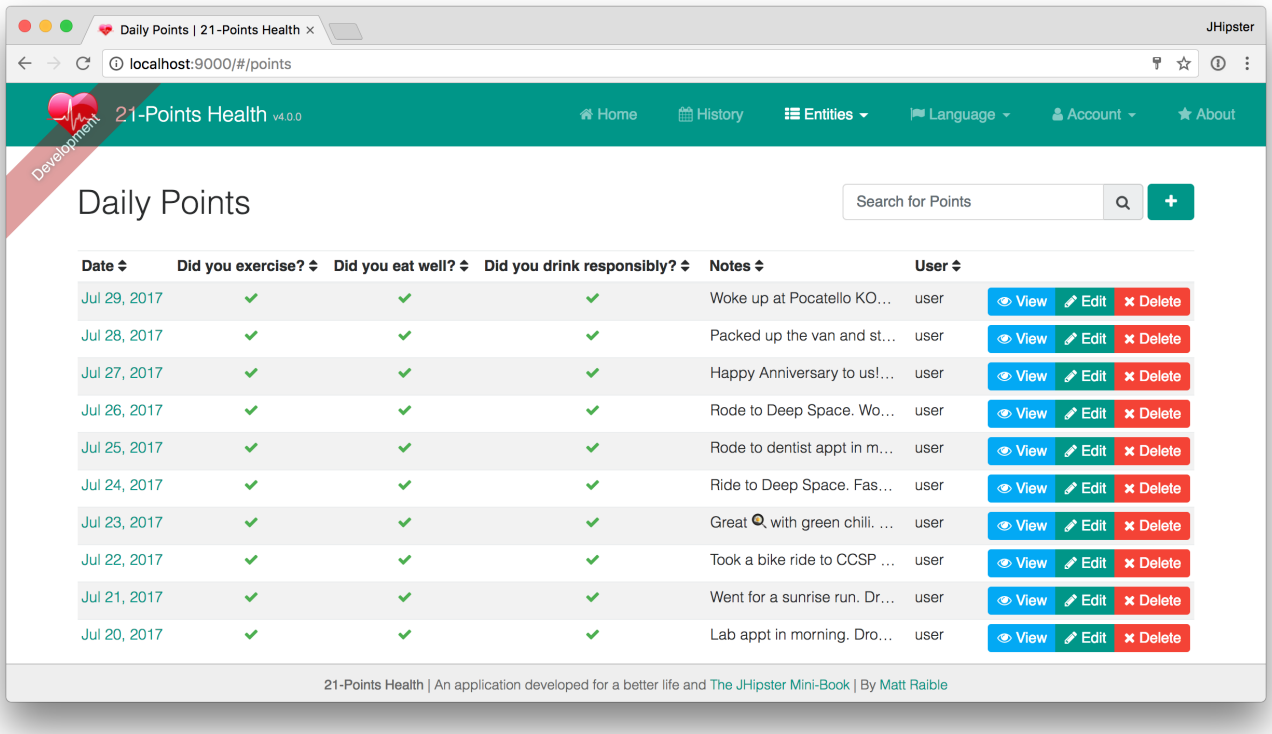

Figure 5. Default Daily Points list after UI improvements

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 a using ng-bootstrap's tooltip directive. The

`jhiTranslate` directive you see in the button tooltips is provided by JHipster's Angular library.

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

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

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

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

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

Similarly, I changed the input fields to checkboxes in `points-dialog.component.html`.

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

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

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

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

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

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

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

After making this change, you can see that the "Add Points" screen is starting to look like the UI mockup I created.

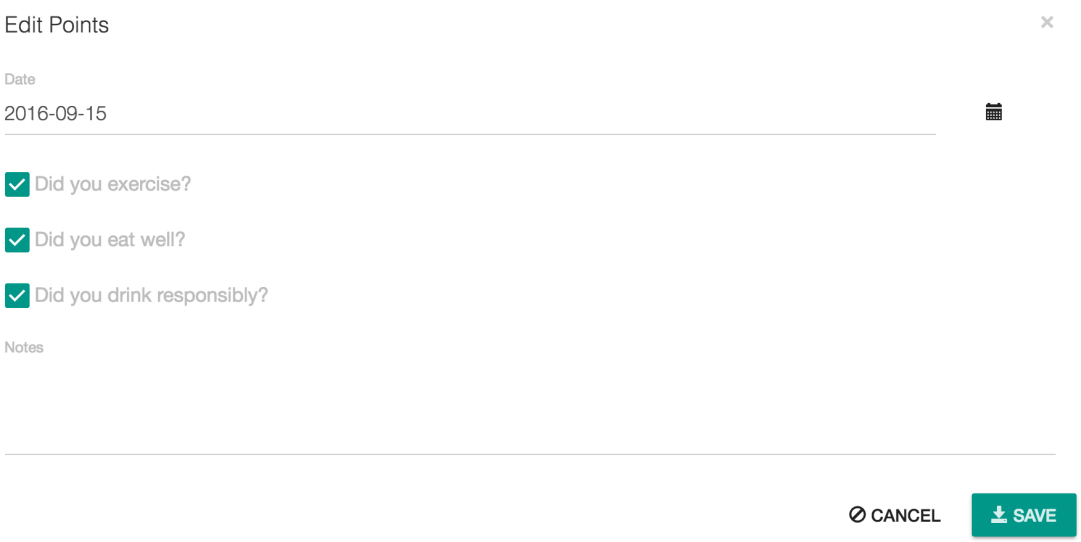

Figure 6. Add Points dialog

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

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

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

Hide user selection from non-admin users

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

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

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

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

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

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

     private final PointsSearchRepository pointsSearchRepository;

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

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

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

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

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

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

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

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

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

+    @Autowired
+    private UserRepository userRepository;

     ...

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

     private Points points;

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

     ...

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

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

         ....
     }
}
```

List screen should show only user's data

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

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

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

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

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

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

Ordering by date

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

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

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

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

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

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

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

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

+    // Create security-aware mockMvc
+    restPointsMockMvc = MockMvcBuilders
+        .webAppContextSetup(context)
+        .apply(springSecurity())
+        .build();
+
     // Get all the points
-    restPointsMockMvc.perform(get("/api/points?sort=id,desc"))
+    restPointsMockMvc.perform(get("/api/points?sort=id,desc")
+            .with(user("admin").roles("ADMIN")))
             .andExpect(status().isOk())
```

Implementing the UI mockup

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

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

I started by reusing the dialogs for entering data that JHipster had created for me. I invoked the dialogs 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]="['/', { outlets: { popup: ['points-new'] } }]"
    class="btn btn-primary m-0 mb-1 text-white">Add Points</a>
```

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

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

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

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

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

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

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

Points this week

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

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

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

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

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

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

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

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

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

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

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

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

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

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

import java.time.LocalDate;

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

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

    public Integer getPoints() {
        return points;
    }

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

    public LocalDate getWeek() {
        return week;
    }

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

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

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

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

```
List<Points> findAllByDateBetween(LocalDate firstDate, LocalDate secondDate);
```

From there, it was just a matter of calculating the beginning and end of the current week and processing the data in `PointsResource.java`.

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

```
/**
 * GET  /points : get all the points for the current week.
 */
@GetMapping("/points-this-week")
@Timed
public ResponseEntity<PointsPerWeek> getPointsThisWeek() {
    // Get current date
    LocalDate now = LocalDate.now();
    // Get first day of week
    LocalDate startOfWeek = now.with(DayOfWeek.MONDAY);
    // Get last day of week
    LocalDate endOfWeek = now.with(DayOfWeek.SUNDAY);
    log.debug("Looking for points between: {} and {}", startOfWeek, endOfWeek);

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

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

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

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

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

```
thisWeek(): Observable<ResponseWrapper> {
    return this.http.get('api/points-this-week')
        .map((res: any) => this.convertResponse(res));
}
```

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

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

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

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

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

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

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

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

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

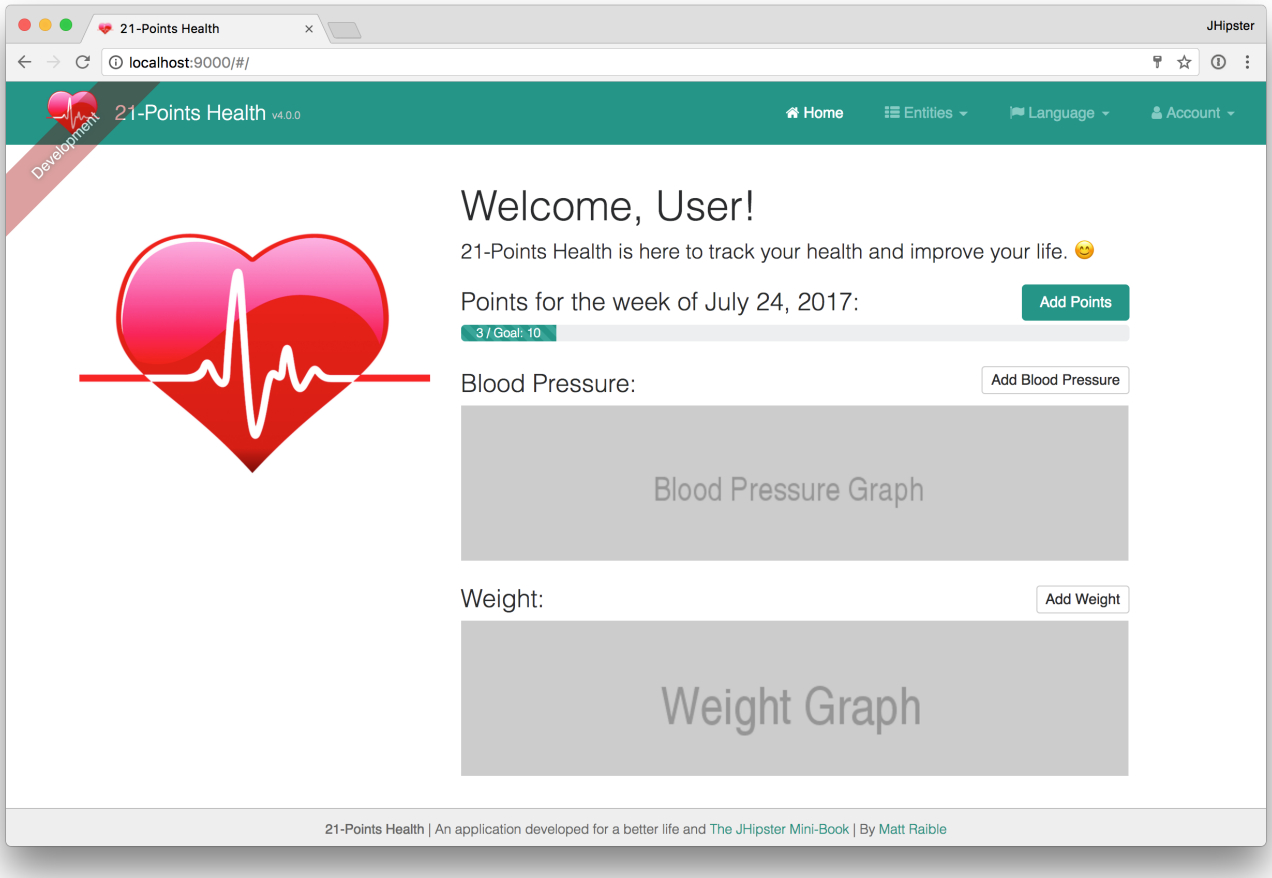

Figure 7. Progress bar for points this week

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

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

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

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

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

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

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

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

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

```
user(): Observable<Preferences> {
    return this.http.get('api/my-preferences').map((res: Response) => {
        return res.json();
    });
}
```

In `home.component.ts`, I added the `PreferencesService` as a dependency and set the preferences in a local

`preferences` variable so the HTML template could read it. I also added a listener for `preference` updates and logic to calculate the background color of the progress bar.

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

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

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

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

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

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

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

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

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

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

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

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

```
<a [routerLink]="['/', { outlets: { popup: 'preferences/'+ preferences.id + '/edit'} }]"
   class="pull-right" jhiTranslate="home.link.preferences">Edit Preferences</a>
```

Blood pressure and weight for the last 30 days

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

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

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

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

    // last month
    bloodPressure = new BloodPressure(firstDayOfLastMonth, 130, 90, user);
    bloodPressureRepository.saveAndFlush(bloodPressure);
    bloodPressure = new BloodPressure(firstDayOfLastMonth.plusDays(11), 135, 85, user);
    bloodPressureRepository.saveAndFlush(bloodPressure);
    bloodPressure = new BloodPressure(firstDayOfLastMonth.plusDays(23), 130, 75, user);
    bloodPressureRepository.saveAndFlush(bloodPressure);
}

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

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

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

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

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

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

```
public class BloodPressureByPeriod {
    private String period;
    private List<BloodPressure> readings;

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

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

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

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

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

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

```
/**
 * GET  /bp-by-days : get all the blood pressure readings by last x days.
 */
@RequestMapping(value = "/bp-by-days/{days}")
@Timed
public ResponseEntity<BloodPressureByPeriod> getByDays(@PathVariable int days) {
    ZonedDateTime rightNow = ZonedDateTime.now();
    ZonedDateTime daysAgo = rightNow.minusDays(days);

    List<BloodPressure> readings = bloodPressureRepository
.findAllByTimestampBetweenOrderByTimestampDesc(daysAgo, rightNow);
    BloodPressureByPeriod response = new BloodPressureByPeriod("Last " + days + " Days",
filterByUser(readings));
    return new ResponseEntity<>(response, HttpStatus.OK);
}

private List<BloodPressure> filterByUser(List<BloodPressure> readings) {
    Stream<BloodPressure> userReadings = readings.stream()
        .filter(bp -> bp.getUser().getLogin().equals(SecurityUtils.getCurrentLogin()));
    return userReadings.collect(Collectors.toList());
}
```

Filtering by method

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

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

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

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

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

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

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

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

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

```
last30Days(): Observable<BloodPressure> {
    return this.http.get('api/bp-by-days/30').map((res: Response) => {
        const jsonResponse = res.json();
        this.convertItemFromServer(jsonResponse);
        return jsonResponse;
    });
}
```

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

Charts of the last 30 days

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

```
yarn add ng2-nvd3
```

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

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

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

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

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

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

```
declare const d3, nv: any;

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

    static getChartConfig() {
        const today = new Date();
        const priorDate = new Date().setDate(today.getDate() - 30);
        return {
            chart: {
                type: 'lineChart',
                height: 200,
                margin: {
```

```
                    top: 20,
                    right: 20,
                    bottom: 40,
                    left: 55
                },
                x(d) {
                    return d.x;
                },
                y(d) {
                    return d.y;
                },
                useInteractiveGuideline: true,
                dispatch: {},
                xAxis: {
                    axisLabel: 'Dates',
                    showMaxMin: false,
                    tickFormat(d) {
                        return d3.time.format('%b %d')(new Date(d));
                    }
                },
                xDomain: [priorDate, today],
                yAxis: {
                    axisLabel: '',
                    axisLabelDistance: 30
                },
                transitionDuration: 250
            },
            title: {
                enable: true
            }
        };
    }
}
```

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

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

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

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

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

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

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

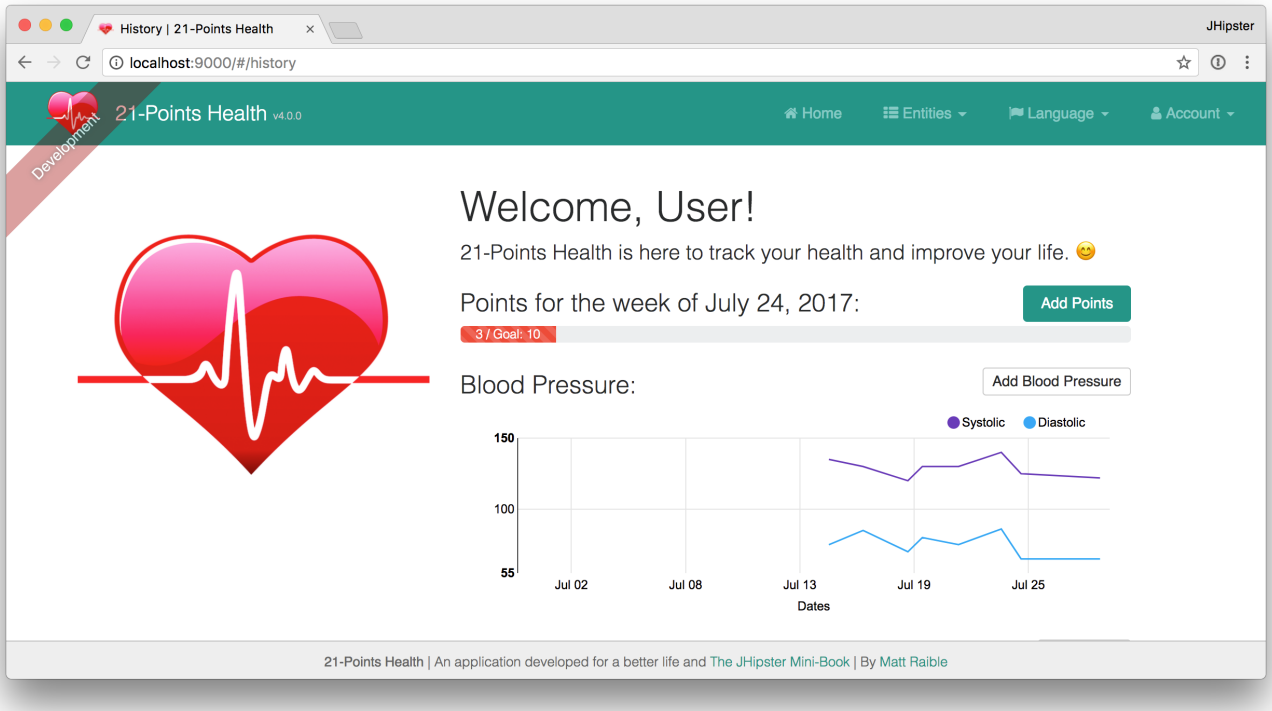

Figure 8. Chart of blood pressure during the last 30 days

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 had produced. You can see from the graph below that I only had to

write 1,521 lines of code. JHipster did the rest for me, generating 93.4% of the code in my project!

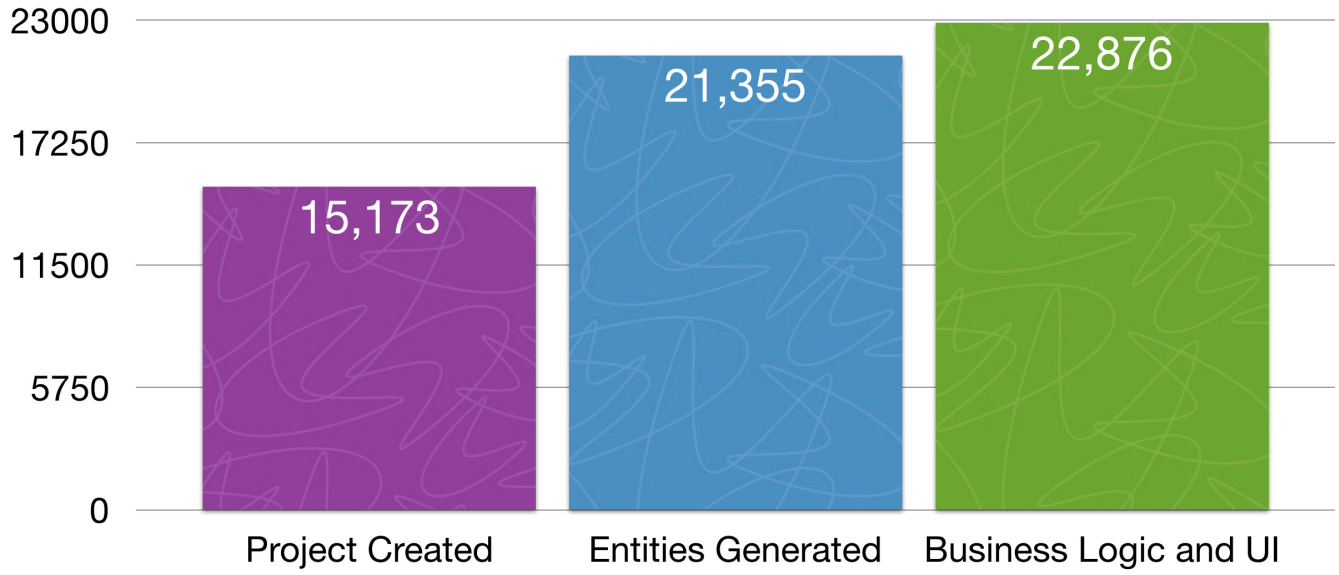

Figure 9. Project lines of code

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

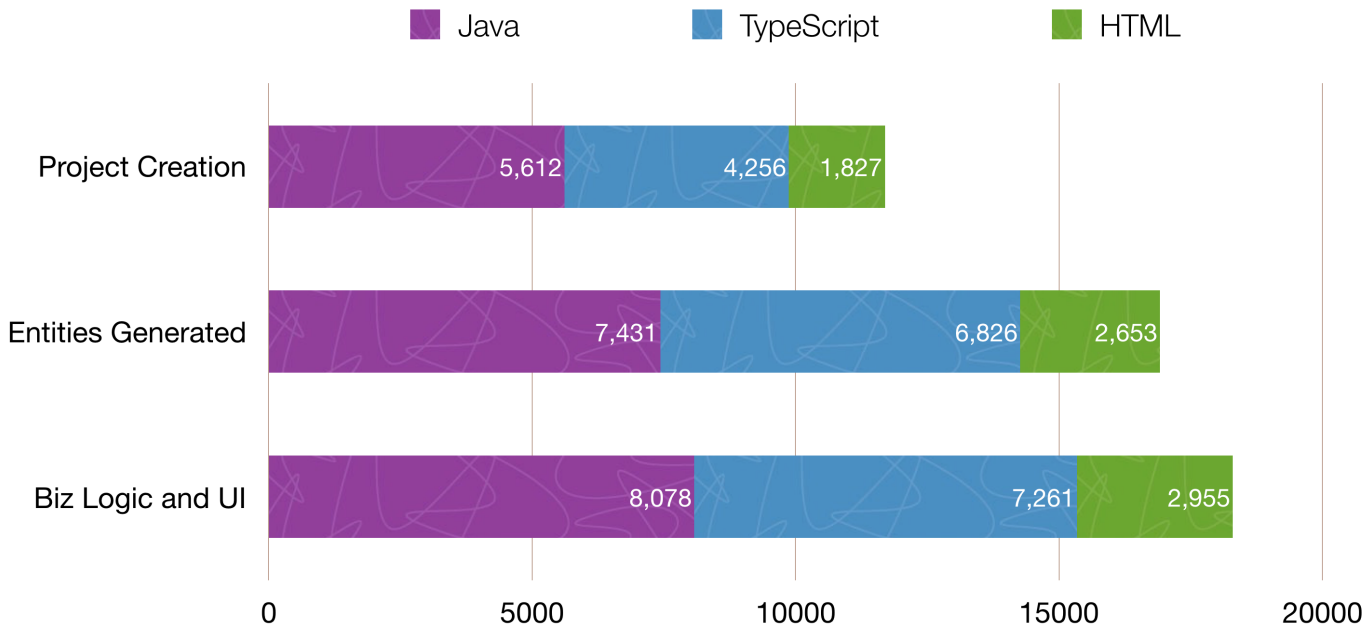

Figure 10. Project lines of code by language

The amount of code I had to write in each language was 647 lines of Java, 435 lines of TypeScript, and 302 lines of HTML. The other 137 lines were Sass (63), JSON (49), CSS (12), YAML (7), and Markdown (6).

Wahoo! Thanks, JHipster!

Testing

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

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

I did not write any tests for this project's UI because I was in a time crunch and I was able to visually verify that things worked as I wanted. I plan to write unit and integration tests when I find the time, but didn't think they were necessary for the MVP.

Deploying to Heroku

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

```
./gradlew -Pprod bootRepackage
```

The command looks similar when using Maven.

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

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

To upload 21-Points Health, I logged in to my Heroku account. I already had the Heroku CLI installed.

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

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

I ran `yo jhipster:heroku` as recommended in the Deploying to Heroku documentation. I tried using the name "21points" for my application when prompted.

```
$ yo jhipster:heroku
  Heroku configuration is starting
  ? Name to deploy as: 21points
  ? On which region do you want to deploy ? us

  Using existing Git repository

  Installing Heroku CLI deployment plugin

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

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

This failed too, because of an issue with JHipster 4.5.2. JHipster 4.5.3 fixes this issue, so I upgraded using the upgrade sub-generator.

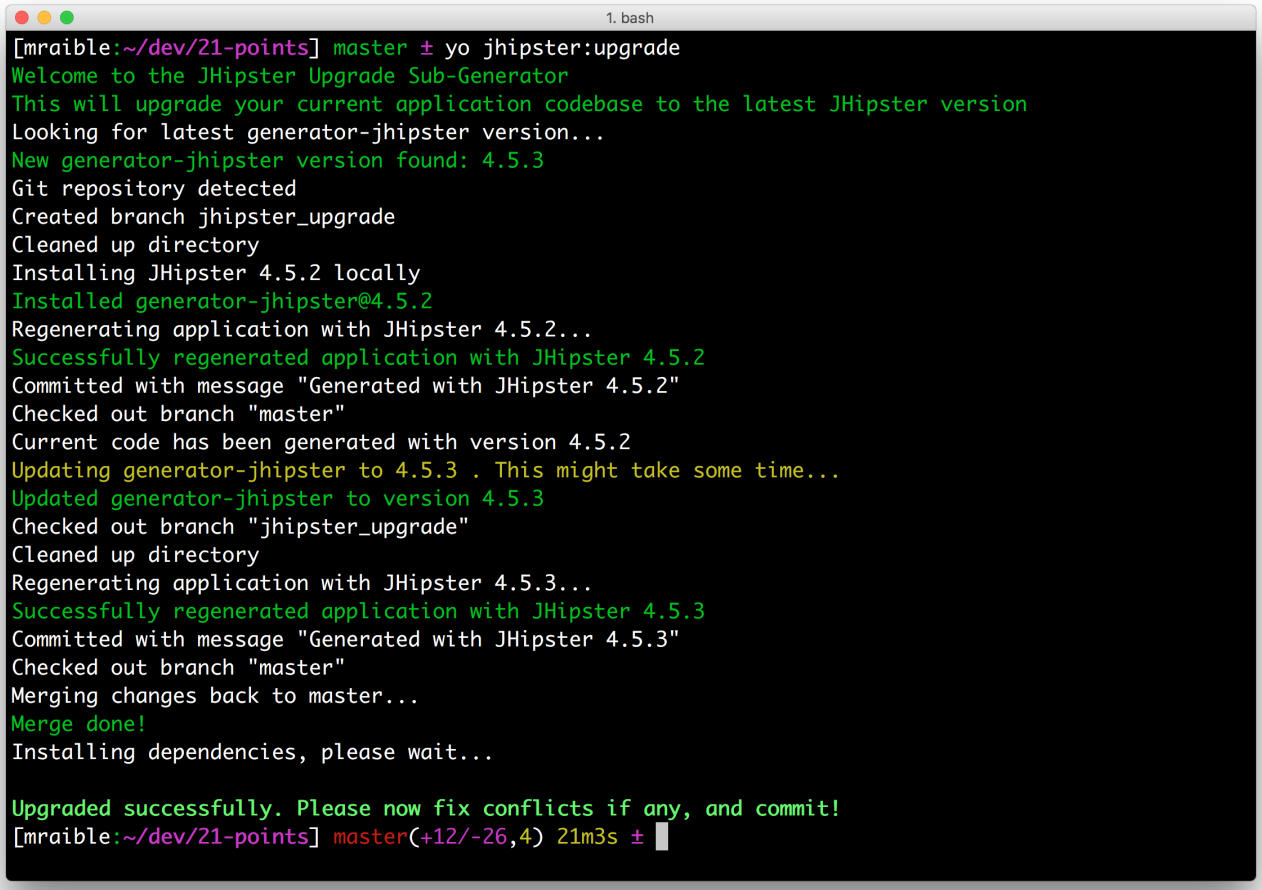

Figure 11. JHipster Upgrade

After the command finished (in just over 21 minutes because I had a slow connection), I committed changes to `yarn.lock` and ran `git push`. If you're reading this and notice that 21-Points Health is using a version newer than 4.5.3, it's likely because I used this handy sub-generator again.

After upgrading, I tried to deploy to Heroku once more.

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

  Using existing Git repository

  Heroku CLI deployment plugin already installed

  Creating Heroku application and setting up node environment
  heroku create health-by-points
  https://health-by-points.herokuapp.com/ | https://git.heroku.com/health-by-points.git

  Provisioning addons
  Created heroku-postgresql --as DATABASE

  Creating Heroku deployment files
     create src/main/resources/config/bootstrap-heroku.yml
     create src/main/resources/config/application-heroku.yml
     create Procfile
     create gradle/heroku.gradle
   conflict build.gradle
  ? Overwrite build.gradle? overwrite this and all others
      force build.gradle

  Building application
...

BUILD SUCCESSFUL
Total time: 3 mins 18.313 secs

Deploying application

Uploading your application code.
This may take several minutes depending on your connection speed...
Uploading twenty-one-points-0.0.1-SNAPSHOT.war
```

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.

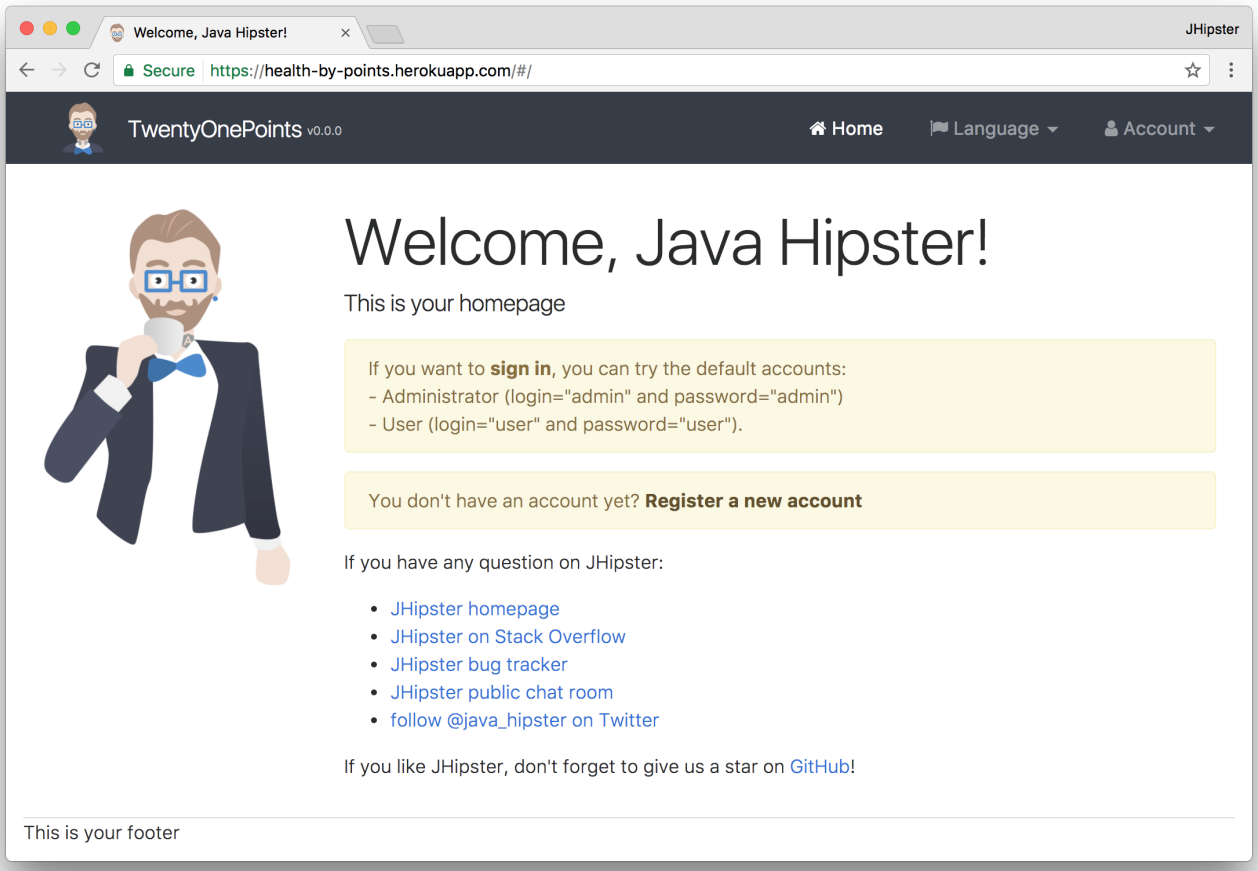

Figure 12. First deployment to Heroku

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. However, after generating entities and adding more code to the project, I found some issues. First of all, I learned that after the initial setup, you can redeploy your application using heroku-cli-deploy. Use the following command to install this plugin.

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

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

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

With Maven, the commands look slightly different:

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

I ran the deployment command after generating all my entities and it looked like everything worked just fine.

```
heroku war:deploy build/libs/*war --app health-by-points
Uploading twenty-one-points-0.0.1-SNAPSHOT.war
-----> Packaging application...
       - app: health-by-points
       - including: webapp-runner.jar
       - including: build/libs/twenty-one-points-0.0.1-SNAPSHOT.war
-----> Creating build...
       - file: slug.tgz
       - size: 73MB
-----> Uploading build...
       - success
-----> Deploying...
remote:
remote: -----> heroku-deploy app detected
remote: -----> Installing OpenJDK 1.8... done
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:        Done: 121.6M
remote: -----> Launching...
remote:        Released v8
remote:        https://health-by-points.herokuapp.com/ deployed to Heroku
remote:
-----> Done
```

I tailed my log files with `heroku logs --tail` to make sure everything started up okay.

It is possible that you'll see the following error on startup.

```
Error R10 (Boot timeout) -> Web process failed to bind to $PORT within 60 seconds of
launch
```

If this happens, create a support ticket at https://help.heroku.com/ and ask to have your application's allowed timeout increased to 120 seconds.

If you need to reset your Postgres database on Heroku, you can do so my logging into https://dashboard.heroku.com. Click on your application name > Add-Ons > Heroku Postgres :: Gray and select "Reset Database" from the gear icon in the top right corner.

Elasticsearch on Heroku

Once my application's timeout was increased, it seemed like everything was working. I tried to register a new user and saw the following error message in my logs.

```
2017-07-29T21:20:57.222878+00:00 app[web.1]: 2017-07-29 21:20:57.222 ERROR 4 ---
[ost-startStop-1] .d.e.r.s.AbstractElasticsearchRepository : failed to load elasticsearch
nodes :
 org.elasticsearch.client.transport.NoNodeAvailableException: None of the configured
nodes are available:
 [{#transport#-1}{localhost}{127.0.0.1:9300}]
```

I searched for an Elasticsearch add-on for Heroku and found Bonsai Elasticsearch. Its cheapest plan cost $10/month. I also found Elasticsearch for $67/month. Since I didn't want to pay for anything right away, I decided to configure Elasticsearch to use an in-memory store like it did in development. (I later discovered that Searchbox Elasticsearch offers a free plan.) I updated my `application-prod.yml` file to use Heroku's ephemeral filesystem.

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

```
# Configure prod to use ElasticSearch in-memory.
# http://stackoverflow.com/questions/12416738/how-to-use-herokus-ephemeral-filesystem
data:
    elasticsearch:
        cluster-name:
        cluster-nodes:
        properties:
            path:
              logs: /tmp/elasticsearch/log
              data: /tmp/elasticsearch/data
```

Mail on Heroku

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

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

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

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

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

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

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

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

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

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

Monitoring and analytics

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

src/main/webapp/index.html

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

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

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

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

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

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

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 `yo jhipster:ci-cd` and watched the magic happen.

```
$ yo jhipster:ci-cd
[Beta] Welcome to the JHipster CI/CD Sub-Generator
? What CI/CD pipeline do you want to generate? Jenkins pipeline
? Jenkins pipeline: what tasks/integrations do you want to include?
? Deploy to heroku? In Jenkins pipeline
   create Jenkinsfile
   create src/main/docker/jenkins.yml
   create src/main/resources/idea.gdsl
```

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

- Perform the build in a Docker container.
- Analyze code with Sonar.
- Send build status to Gitlab.
- Build and publish a Docker image.

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

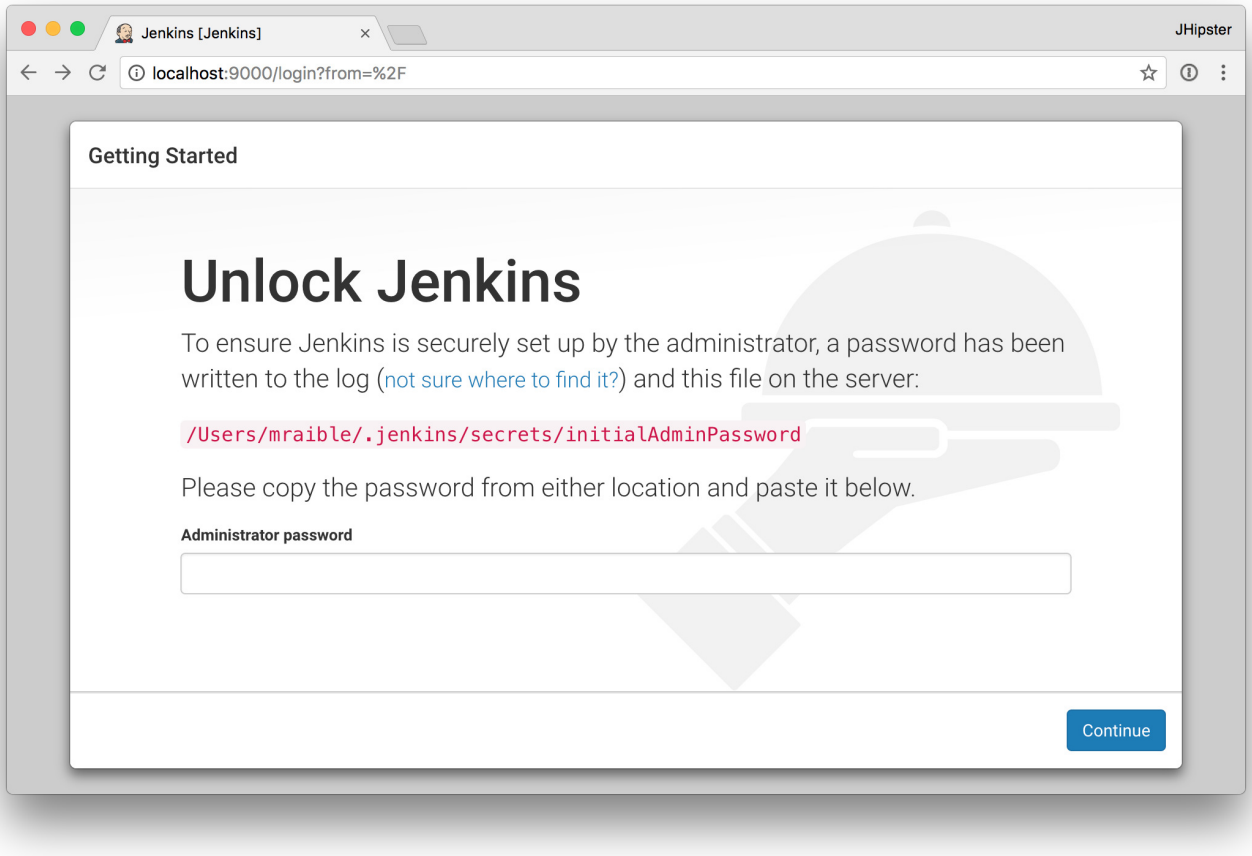

Figure 13. Unlock Jenkins

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

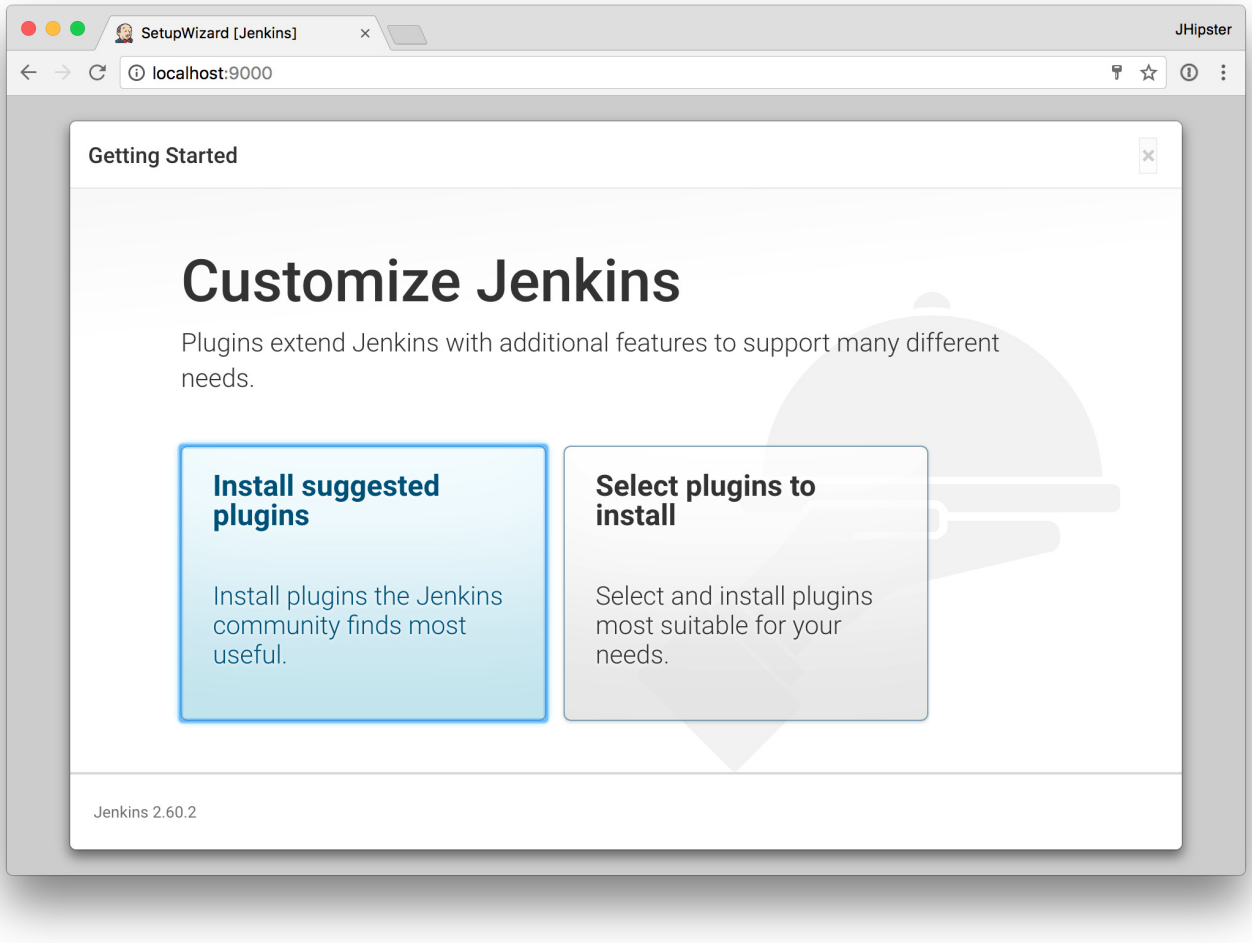

Figure 14. Customize Jenkins

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

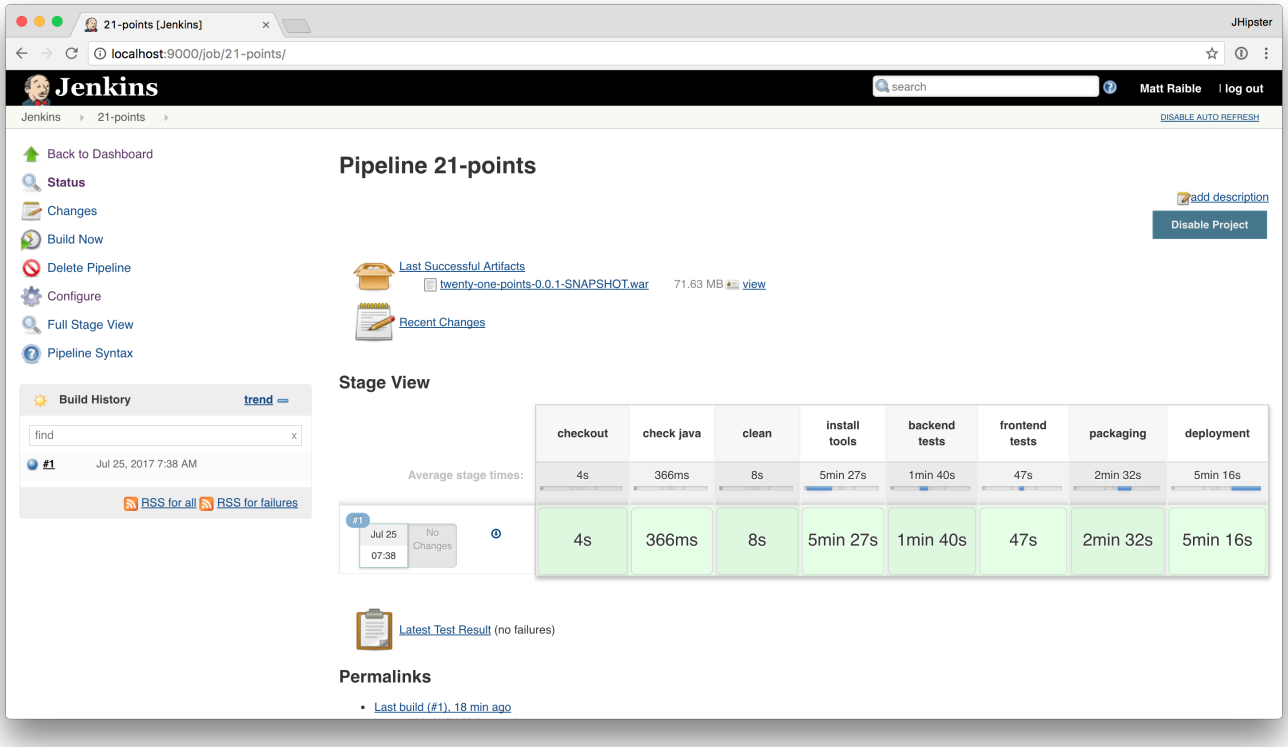

Figure 15. Jenkins build #1

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

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

Jenkinsfile

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

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

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

    stage('clean') {
        sh "chmod +x gradlew"
        sh "./gradlew clean --no-daemon"
    }
```

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

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

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

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

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

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

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

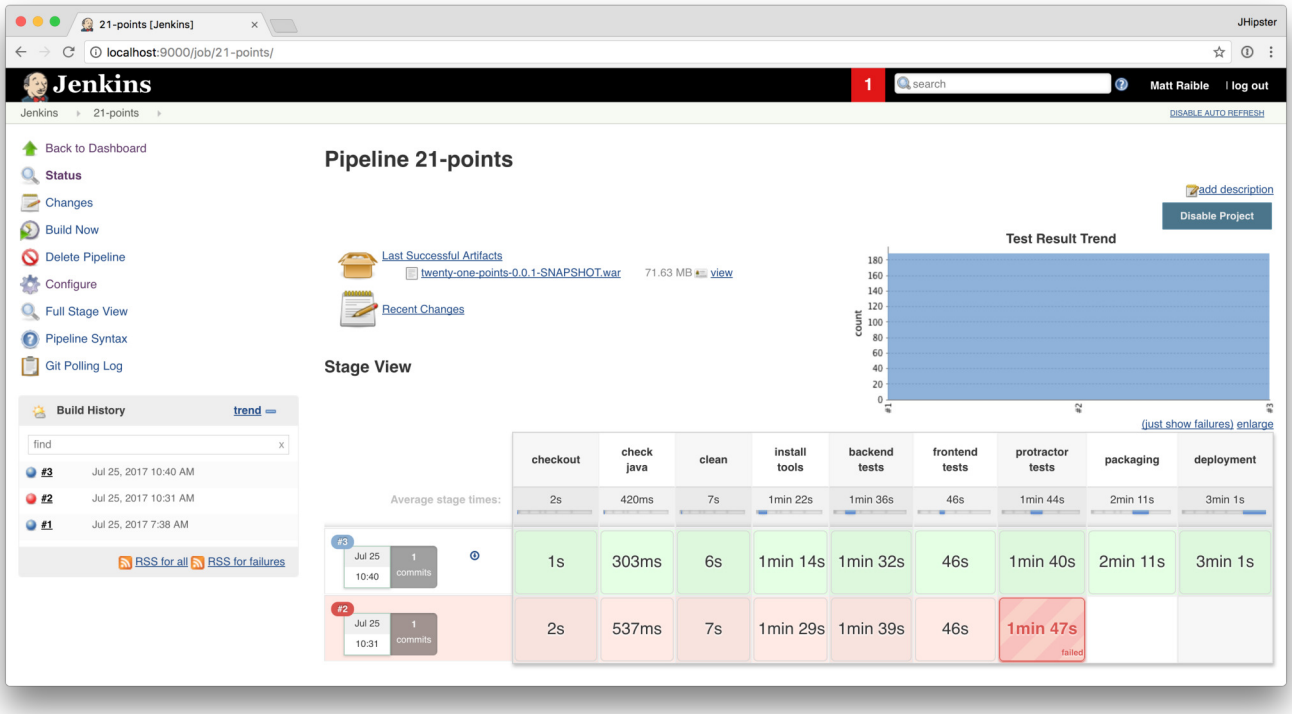

Figure 16. Jenkins success!

The reason job #2 didn't pass in the screenshot above is because I initially slept 45 seconds before running Protractor tests. This was not enough time for Spring Boot to start up. Increasing it to 60 seconds solved the problem.

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.

Source code

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

Upgrading 21-Points Health

After I finished developing the MVP of 21-Points Health with JHipster 4.5.3, I wanted to upgrade it to the latest release. I tried using the upgrade sub-generator, but found that it was broken in 4.5.3 and only fixed in 4.5.6. Therefore, I manually upgraded 4.5.3 to 4.6.2 using SmartSynchronize. You can see what changed when I upgraded by taking a look at the commit on GitHub.

If you're reading this and notice that 21-Points Health is using a version newer than 4.6.2, it's likely because I upgraded again.

If you have a JHipster 3.x app that was written in AngularJS, you might try these instructions for upgrading.

Summary

This section showed you how I created a health-tracking web application with JHipster. It walked you through upgrading to the latest release of JHipster and how to generate code with `yo 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 dialogs on different pages, how to add methods to client services, and how to manipulate data to display pretty charts.

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

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

PART TWO

JHipster's UI components

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

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

Angular

JHipster supports two UI frameworks: AngularJS and Angular. It will also support React by the end of 2017. In the first two versions of this book, I showed you how to use AngularJS. In this version, I'll show you how to use Angular. You can see from the following graphs that Angular and React are the most popular among JavaScript frameworks.

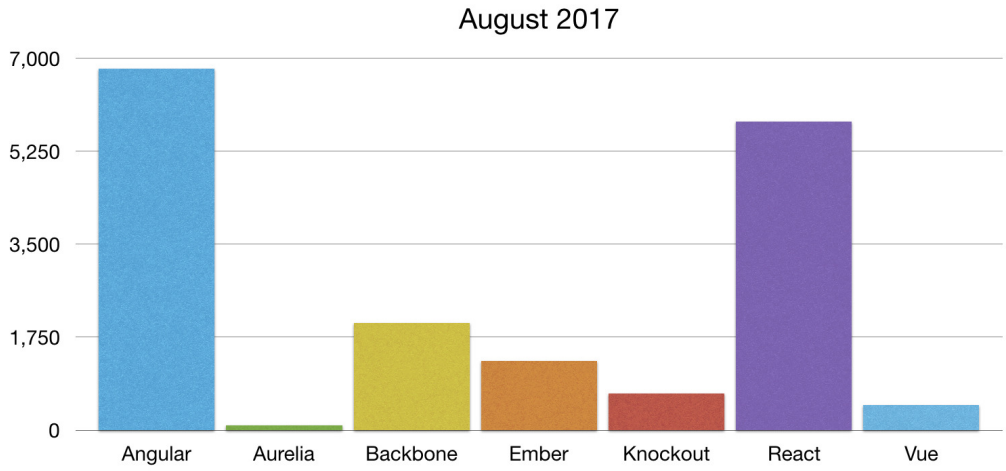

Figure 17. Jobs on Indeed, August 2017

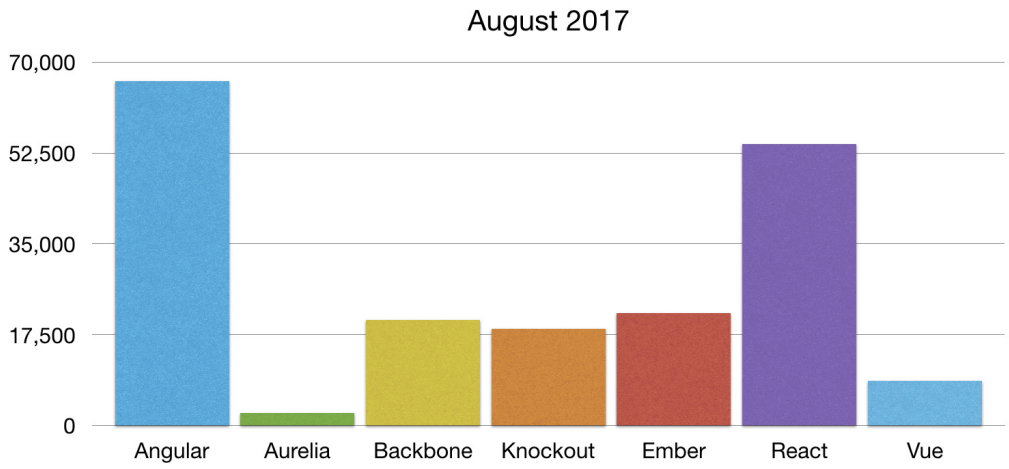

Figure 18. Stack Overflow Tags, August 2017

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

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

History

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

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

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

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

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

Angular 2.0 was released September 2016. Angular 4.0 was released March 2017. JHipster 4.6.0 was released July 6, 2017 and contains production-ready Angular support. Prior to that, JHipster's Angular

support was considered beta quality.

The Angular project is hosted at https://angular.io.

Basics

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

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

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

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

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

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

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

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

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

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

platformBrowserDynamic().bootstrapModule(AppModule);
```

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

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

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

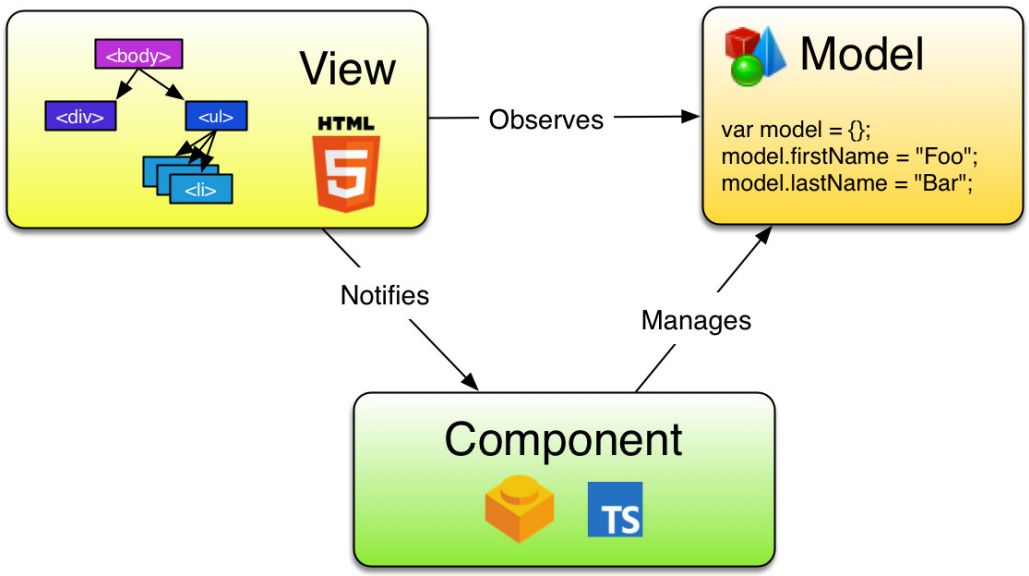

Figure 19. MVC in Angular

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

SearchService

```
import { Injectable } from '@angular/core';
import { Http, Response } from '@angular/http';
import 'rxjs/add/operator/map';
import { Observable } from 'rxjs';

@Injectable()
export class SearchService {

  constructor(private http: Http) {}

  search(term): Observable<any> {
    return this.http.get(`/api/search/${term}`).map((res: Response) => res.json());
  }
}
```

An associated `SearchComponent` can be used to display the results from this service. Notice how you can use constructor injection to get a reference to the service.

SearchComponent

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

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

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

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

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

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

AppModule

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

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

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

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

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

app.component.html

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

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

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

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

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

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

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

.controller('SearchController', function ($scope, SearchService) { ①
    $scope.search = function () { ②
        console.log("Search term is: " + $scope.term); ③
        SearchService.query($scope.term).then(function (response) {
            $scope.searchResults = response.data;
        });
    };
})
```

① To inject `SearchService` into `SearchComponent`, simply add it as a parameter to the component's constructor. Note that the service has to be listed as a provider in `@Component`, or in `AppModule`. 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 `[(ngModule)]` 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 Yarn.

```
yarn add global @angular/cli
```

Then generate a new application using `ng new`.

```
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.

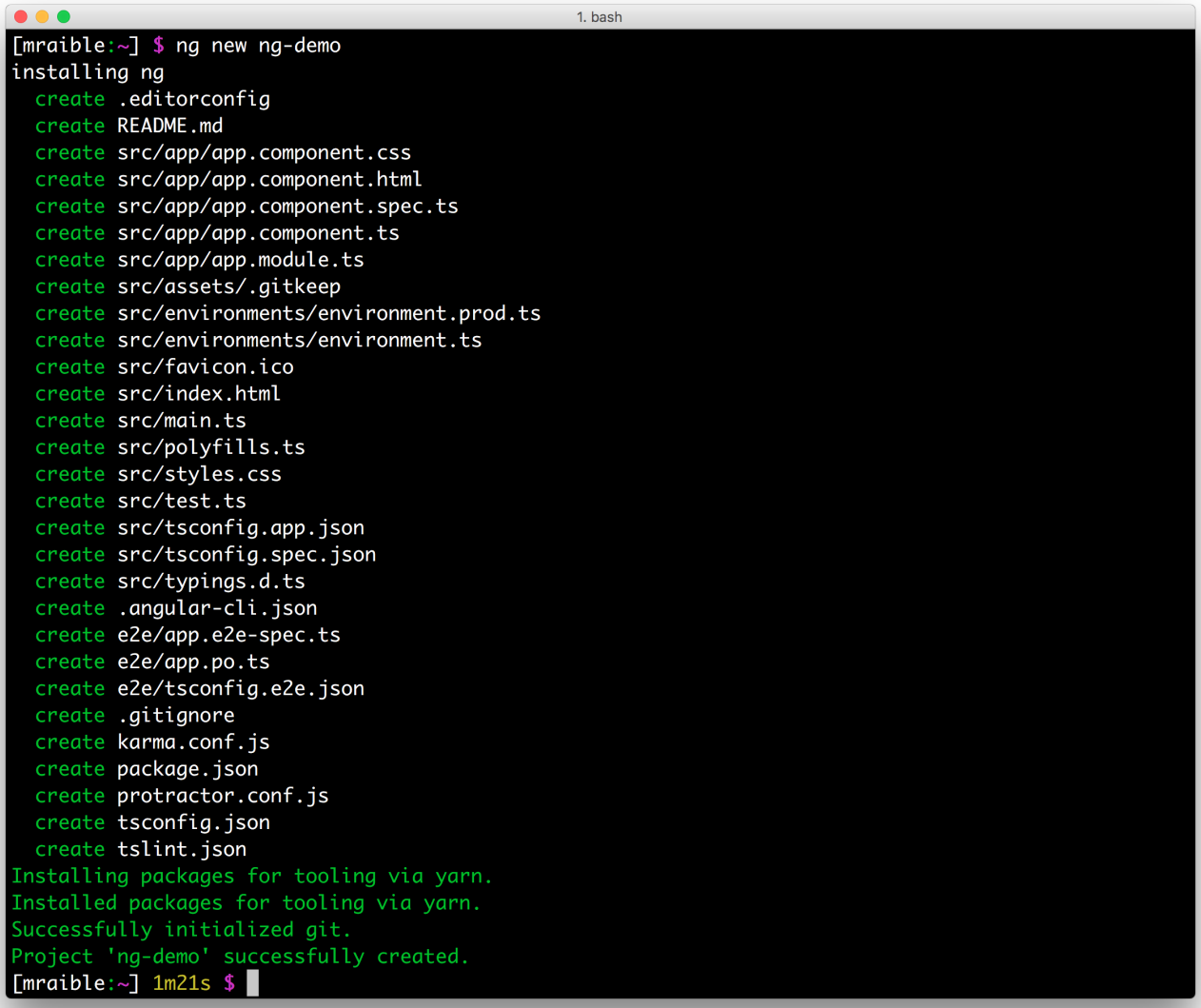

Figure 20. Creating a new Angular project with Angular CLI

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

```
$ ng g s search
installing service
  create src/app/search.service.spec.ts
  create src/app/search.service.ts
  WARNING Service is generated but not provided, it must be provided to be used
```

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

```
$ ng g c search
installing component
  create src/app/search/search.component.css
  create src/app/search/search.component.html
  create src/app/search/search.component.spec.ts
  create src/app/search/search.component.ts
  update src/app/app.module.ts
```

Does your API return data like the following?

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

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

```
<table *ngIf="searchResults">
  <thead>
  <tr>
    <th>Name</th>
    <th>Phone</th>
    <th>Address</th>
  </tr>
  </thead>
  <tbody>
  <tr *ngFor="let person of searchResults; let i=index">
    <td><a href="" (click)="onSelect(person); false">{{person.name}}</a></td>
    <td>{{person.phone}}</td>
    <td>{{person.address.street}}<br/>
      {{person.address.city}}, {{person.address.state}} {{person.address.zip}}
    </td>
  </tr>
  </tbody>
</table>
```

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

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

Bootstrap

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

Bootstrap's grid system

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

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

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

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

A basic example of the grid is shown below.

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

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

Figure 21. 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 22. Basic grid on a mobile device

Bootstrap's grid can be used to align and position your application's elements, widgets, and features.

It's helpful to understand a few basics if want to use it effectively.

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

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

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

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

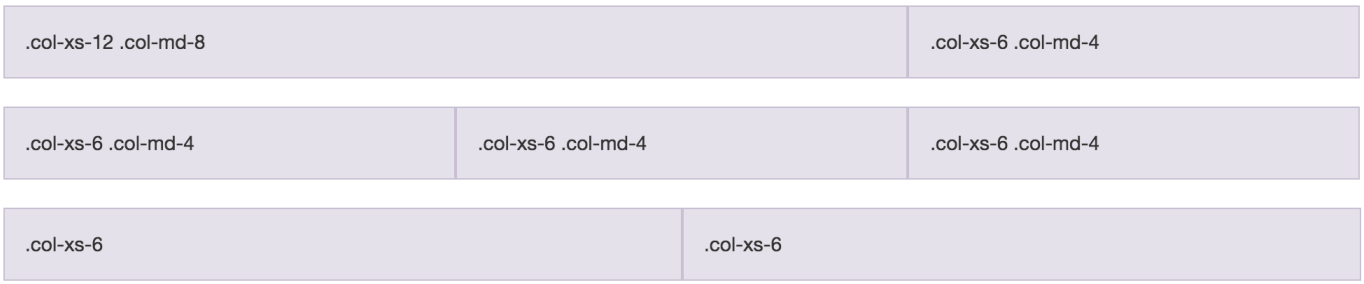

Figure 23. Advanced grid

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) { ... }
```

Responsive utility classes

Bootstrap also includes a number of utility classes that can be used to show and hide elements based on browser size, like `.hidden-[xs|sm|md|lg]-up` and `.hidden-[xs|sm|md|lg]-down`. There are no explicit "show" responsive utility classes; you make an element visible by simply not hiding it at that breakpoint size. You can combine one `.hidden-*-up` class with one `.hidden-*-down` class to show an element only on a given interval of screen sizes. For example, `.hidden-sm-down.hidden-xl-up` shows the element only on medium and large viewports.

Forms

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

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

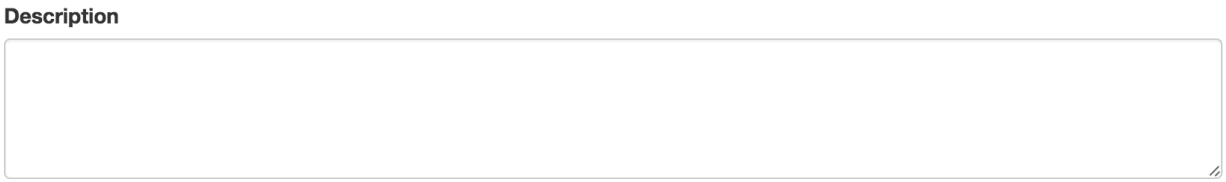

Figure 24. Basic form element

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

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

Figure 25. Form element with validation

CSS

When you add Bootstrap's CSS to a HTML page, the default settings immediately improve the typography. Your `<h1>` and `<h2>` headings become semi-bold and are sized accordingly. Your paragraph margins, body text, and block quotes will look better. If you want to align text in your pages, `text-[left|center|right]` are useful classes. For tables, a `table` class gives them a better look and feel by default.

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

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

Figure 26. Buttons

Components

Bootstrap ships with a number of components included. Some require JavaScript; some only require

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

Some 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.

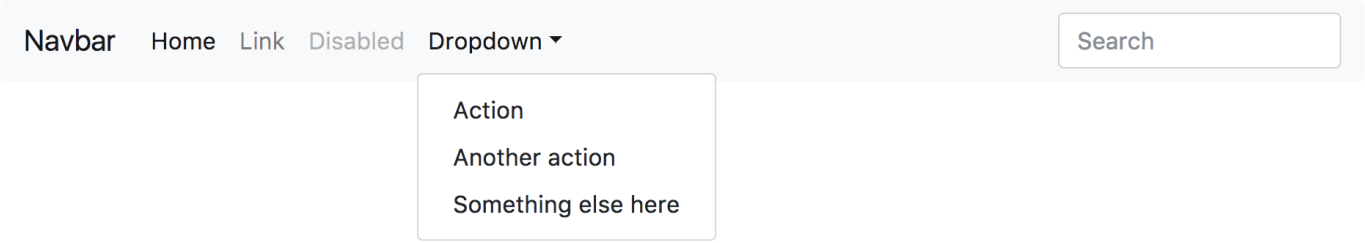

Figure 27. Navbar with dropdown

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

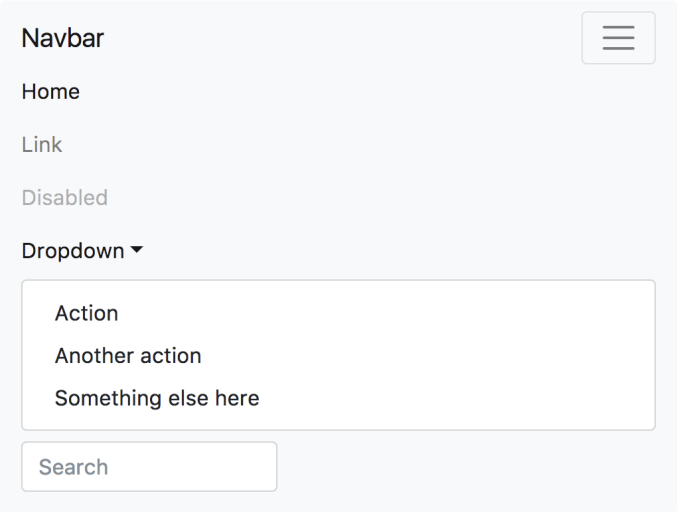

Figure 28. Navbar on mobile

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

```
<nav class="navbar">
    <a class="navbar-brand" href="#">Navbar</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=
"#dropdown">
        <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse">
        <ul class="navbar-nav mr-auto">
            <li class="nav-item active">
                <a class="nav-link" href="#">Home</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#">Link</a>
            </li>
            <li class="nav-item dropdown">
                <a class="nav-link dropdown-toggle" id="dropdown" data-toggle="dropdown"
>Dropdown</a>
                <div class="dropdown-menu" aria-labelledby="dropdown">
                    <a class="dropdown-item" href="#">Action</a>
                    <a class="dropdown-item" href="#">Another action</a>
                    <a class="dropdown-item" href="#">Something else here</a>
                </div>
            </li>
        </ul>
        <form class="form-inline">
            <input class="form-control" type="text" placeholder="Search">
        </form>
    </div>
</nav>
```

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

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

This renders alerts like the following.

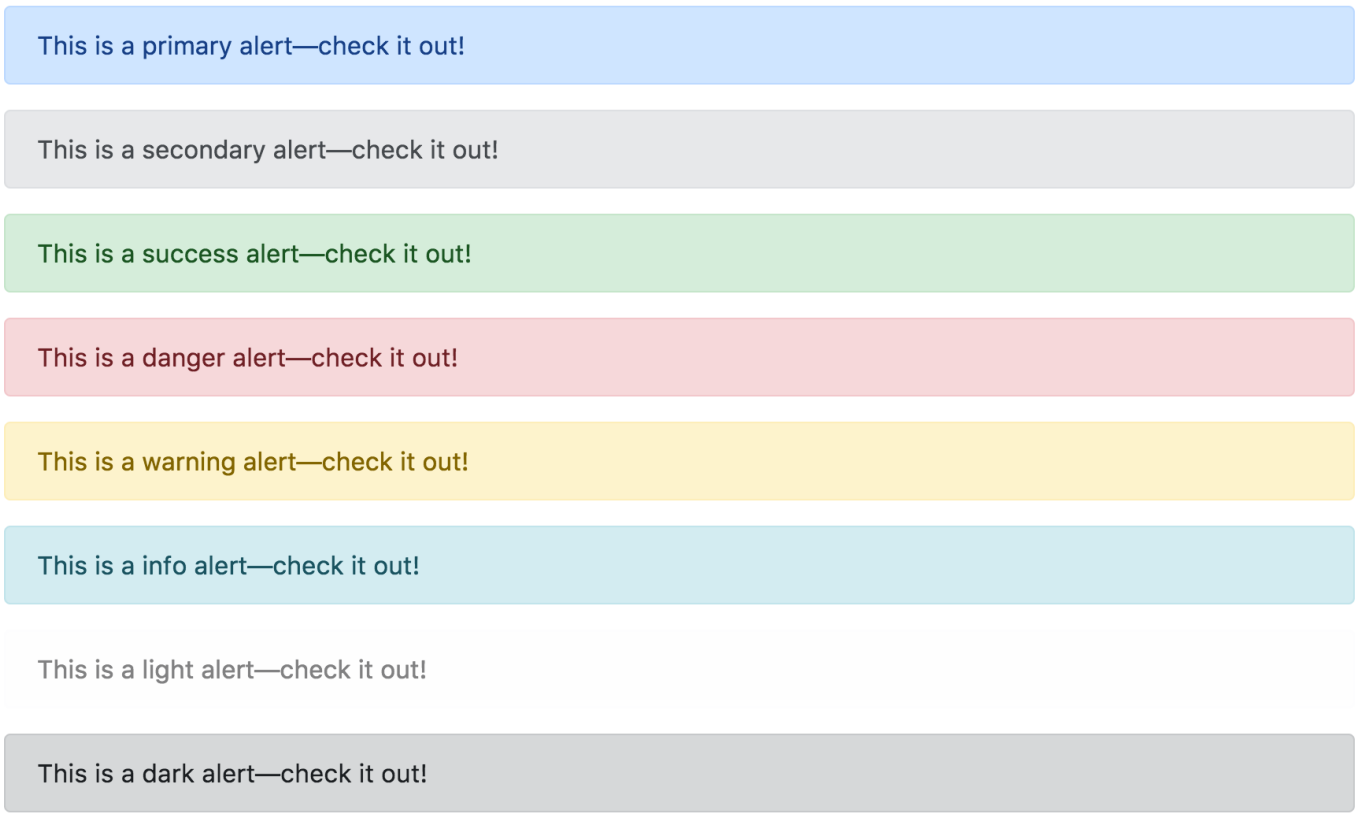

Figure 29. Alerts

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

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

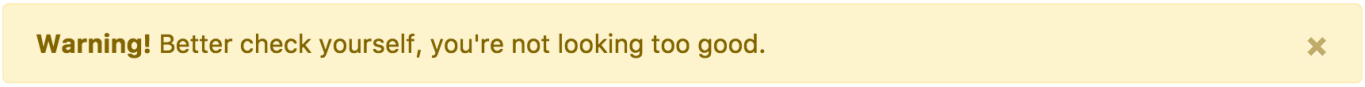

Figure 30. Closeable alert

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

Icons

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

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

Font Awesome has 675 icons and is included by default in JHipster. It is often used to display eye candy on buttons.

```
<button class="btn btn-info"><i class="fa fa-plus"></i> Add</button>
<button class="btn btn-danger"><i class="fa fa-remove"></i> Delete</button>
<button class="btn btn-success"><i class="fa fa-pencil"></i> Edit</button>
```

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

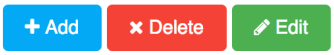

Figure 31. Buttons with icons

Customizing CSS

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

src/main/webapp/scss/vendor.scss

```
/* after changing this file run 'yarn run 'yarn run webpack:build' */
$fa-font-path: '~font-awesome/fonts';

/***************************
put Sass variables here:
eg $input-color: red;
***************************/
// Override Boostrap variables
@import "bootstrap-variables";
// Import Bootstrap source files from node_modules
@import 'node_modules/bootstrap/scss/bootstrap';
@import 'node_modules/font-awesome/scss/font-awesome';
```

JHipster 4.7.0+ includes a `src/main/webapp/content/scss/_bootstrap-variable.scss` file. You can modify

this file to change the default Bootstrap settings like colors, border radius, etc.

Angular and Bootstrap

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

Toastr is a popular non-blocking notification library powered by jQuery. To use something similar in an Angular application, you can choose from several implementations:

- ng2-toastr (265 GitHub stars)
- Angular2-Toaster (203 GitHub stars)
- ngx-toastr (179 GitHub stars)

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

Internationalization (i18n)

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

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

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

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

JHipster allows you to choose a default language and translations when you first create a project. It stores the JSON documents for these languages in `src/main/webapp/i18n`. You can install additional languages using `yo jhipster:languages`. At the time of this writing, JHipster supports 37 languages. You can also add a new language. To set the default language, modify `src/main/webapp/app/shared/shared-common.module.ts` and its `LOCALE_ID` setting.

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

```
providers: [
    JhiLanguageHelper,
    Title,
    {
        provide: LOCALE_ID,
        useValue: 'en'
    },
],
```

Sass

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

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

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

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

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

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

The code above renders the following CSS.

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

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

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

    li {
        display: inline-block;
    }

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

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

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

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

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

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

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

CSS generated from the above code looks as follows.

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

Bootstrap 3.x was written with Less, a CSS pre-processor with similar features to Sass. It uses Sass for the 4.0 version.

JHipster allows you to use Sass by integrating LibSass with node-sass. 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 uses Gulp. Gulp allows you to perform tasks like minification, concatenation, compilation (e.g. from TypeScript/CoffeeScript to JavaScript), unit testing, and more. webpack is a more modern solution that's become very popular for Angular projects and is included under-the-covers in Angular CLI.

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

It has four core concepts:

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

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

webpack.config.js

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

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

module.exports = config;
```

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

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

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

- `webpack.test.js` is the configuration used for running tests, where files are watched and reloaded when changed.

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

WebSockets

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

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

- STOMP WebSocket: STOMP stands for "simple text-oriented messaging protocol".
- SockJS: SockJS provides a WebSocket-like object. If native WebSockets are not available, it falls back to other browser techniques.

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

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

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

import { JhiTrackerService } from '../../shared';

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

    activities: any[] = [];

    constructor(
        private trackerService: JhiTrackerService
    ) {
    }

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

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

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

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

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

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

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

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

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

    constructor(
        private router: Router,
        private authServerProvider: AuthServerProvider,
        private $window: WindowRef,
        private csrfService: CSRFService
    ) {
        this.connection = this.createConnection();
        this.listener = this.createListener();
    }

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

```
            url += '?access_token=' + authToken;
        }
        const socket = new SockJS(url);
        this.stompClient = Stomp.over(socket);
        const headers = {};
        this.stompClient.connect(headers, () => {
            this.connectedPromise('success');
            this.connectedPromise = null;
            this.sendActivity();
            if (!this.alreadyConnectedOnce) {
                this.subscription = this.router.events.subscribe((event) => {
                    if (event instanceof NavigationEnd) {
                        this.sendActivity();
                    }
                });
                this.alreadyConnectedOnce = true;
            }
        });
    }

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

    receive() {
        return this.listener;
    }

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

    subscribe() {
        this.connection.then(() => {
```

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

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

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

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

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

Browsersync

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

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

- Interaction sync: Browsersync mirrors your scroll, click, refresh, and form actions between browsers while you test.
- File sync: Browsers automatically update as you change HTML, CSS, images, and other project files.
- URL history: Browsersync records your test URLs so you can push them back out to all devices with a single click.

- Remote inspector: You can remotely tweak and debug web pages that are running on connected devices.

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

```
{
  "name": "jhipster-book",
  "version": "4.0.0",
  "description": "The JHipster Mini-Book",
  "repository": {
    "type": "git",
    "url": "git@bitbucket.org:mraible/jhipster-book.git"
  },
  "devDependencies": {
    "gulp": "3.9.1",
    "browser-sync": "2.18.13"
  }
}
```

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

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

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

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

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

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

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

```
npm install
```

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

```
gulp
```

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

Summary

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

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

PART THREE

JHipster's API building blocks

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

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

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

Spring Boot

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

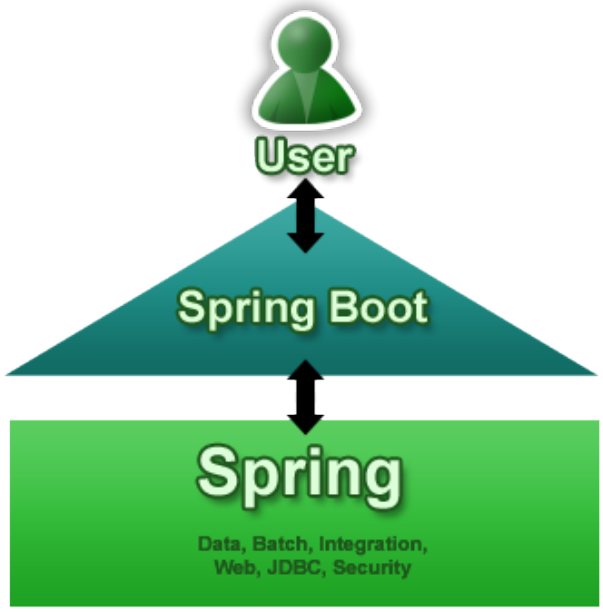

Figure 32. Spring Boot

The primary goals of Spring Boot are:

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

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

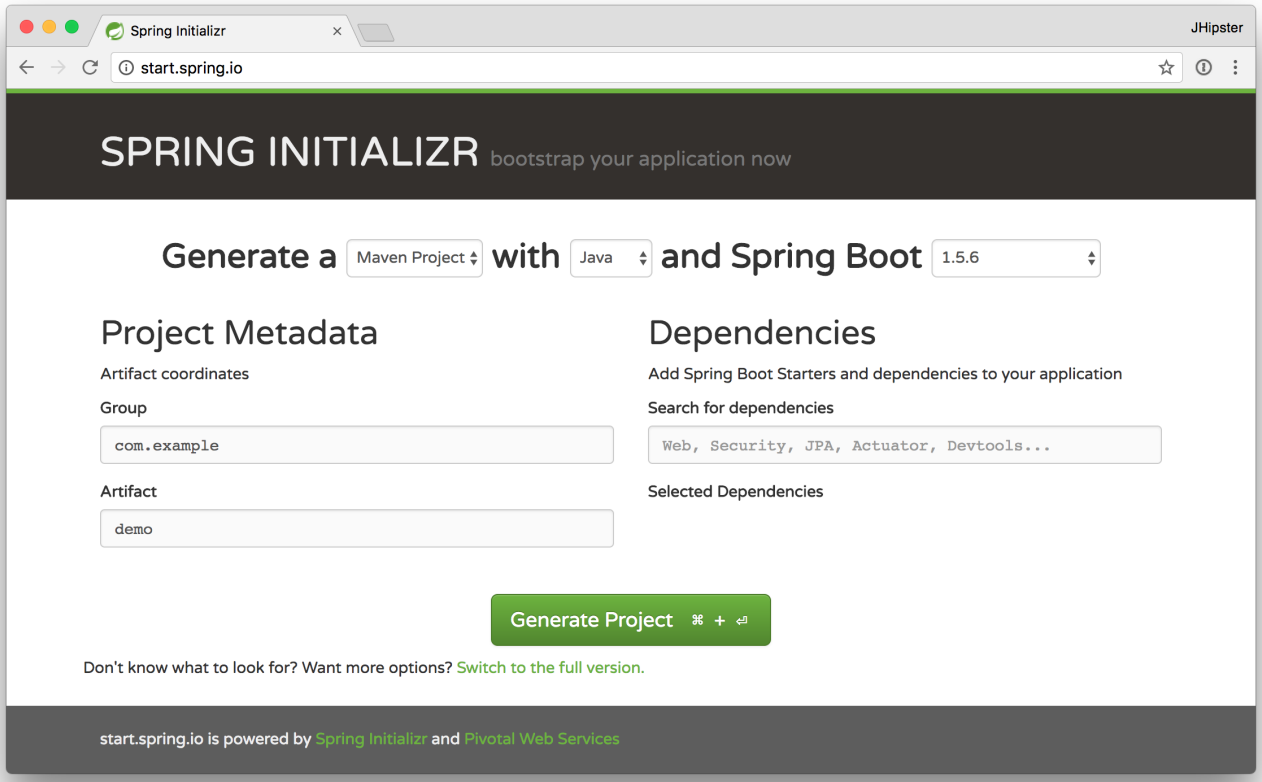

Figure 33. Spring Initializr in a browser

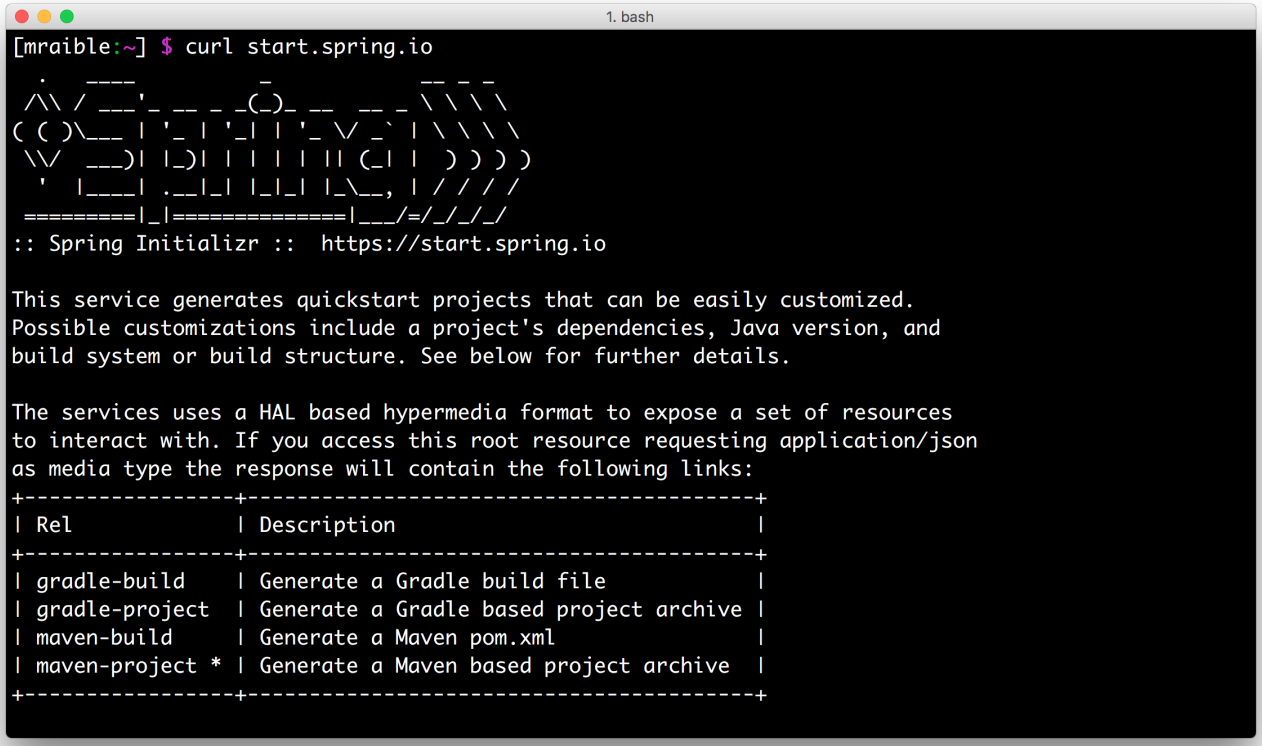

Figure 34. Spring Initializr via `curl`

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

At the bottom of the start.spring.io page, you can also download or install the Spring CLI (also called the Spring Boot CLI). The easiest way to install it is with the following command.

```
curl https://start.spring.io/install.sh | sh
```

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

hello.groovy

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

To compile and run this application, simply type:

```
spring run hello.groovy
```

After running this command, you can see the application at http://localhost:8080. For more information about the Spring CLI, see the Spring Boot documentation.

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

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

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

14 directories, 6 files
```

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

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

```
package com.example.demo;

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

@SpringBootApplication
public class DemoApplication {

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

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

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

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
...

@Entity
class Blog {

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

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

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

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

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

```
import org.springframework.data.jpa.repository.JpaRepository;
....

interface BlogRepository extends JpaRepository<Blog, Long> {}
```

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

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

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

...

@Component
class BlogCommandLineRunner implements CommandLineRunner {

    private BlogRepository repository;

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

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

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

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

src/main/resources/data.sql

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

Start your application with `mvn spring-boot:run` (or right-click → "Run in your IDE") and you should

see this default data show up in your logs.

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

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

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

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.Collection;
...

@RestController
class BlogController {
    private final BlogRespository 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"}]
```

Spring has one of the best track records for hipness in Javaland. It is an essential cornerstone of the solid API foundation that makes JHipster awesome. Spring Boot allows you to create stand-alone Spring applications that directly embed Tomcat, Jetty, or Undertow. It provides opinionated starter

dependencies that simplify your build configuration, regardless of whether you're using Maven or Gradle.

External configuration

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

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

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

Application property files

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

1. a `/config` subdirectory of the current directory,

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

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

More information about Spring Boot's external-configuration feature can be found in Spring Boot's "Externalized Configuration" reference documentation.

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

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

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

Automatic configuration

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

spring-boot-actuator.jar!/META-INF/spring.factories

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.actuate.autoconfigure.AuditAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.CacheStatisticsAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.CrshAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.EndpointAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.EndpointMBeanExportAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.EndpointWebMvcAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.HealthIndicatorAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.InfoContributorAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.JolokiaAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.ManagementServerPropertiesAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.ManagementWebSecurityAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.MetricFilterAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.MetricRepositoryAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.MetricsDropwizardAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.MetricsChannelAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.MetricExportAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.PublicMetricsAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.TraceRepositoryAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.TraceWebFilterAutoConfiguration,\
org.springframework.boot.actuate.cloudfoundry.CloudFoundryActuatorAutoConfiguration

org.springframework.boot.actuate.autoconfigure.ManagementContextConfiguration=\
org.springframework.boot.actuate.autoconfigure.EndpointWebMvcManagementContextConfiguration,\
org.springframework.boot.actuate.autoconfigure.EndpointWebMvcHypermediaManagementContextConfiguration
```

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

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

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

Actuator

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

```
<dependencies>
    <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 a number of REST endpoints. By default, Spring Boot will also expose management endpoints as JMX MBeans under the `org.springframework.boot` domain. Actuator REST endpoints include:

- `/actuator` — Provides a hypermedia-based "discovery page" for the other endpoints. Requires Spring HATEOAS to be on the classpath.
- `/auditevents` — Exposes audit events information for the current application.
- `/autoconfig` — Returns an auto-configuration report that shows all auto-configuration candidates.
- `/beans` — Returns a complete list of all the Spring beans in your application.
- `/configprops` — Returns a list of all `@ConfigurationProperties`.
- `/dump` — Performs a thread dump.
- `/env` — Returns properties from Spring's `ConfigurableEnvironment`.
- `/flyway` — Shows any Flyway database migrations that have been applied.
- `/health` — Returns information about application health.
- `/info` — Returns basic application info.
- `/loggers` — Shows and modifies the configuration of loggers in the application.
- `/liquibase` — Shows any Liquibase database migrations that have been applied.
- `/metrics` — Returns performance information for the current application.
- `/mappings` — Returns a list of all `@RequestMapping` paths.
- `/shutdown` — Shuts the application down gracefully (not enabled by default).
- `/trace` — Returns trace information (by default, the last several HTTP requests).

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-mail
spring-boot-starter-logging
spring-boot-starter-aop
spring-boot-starter-data-jpa
spring-boot-starter-data-elasticsearch
spring-boot-starter-security
spring-boot-starter-web
spring-boot-starter-undertow
spring-boot-starter-thymeleaf
spring-boot-starter-cloud-connectors
spring-boot-starter-test
```

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

Maven versus Gradle

Maven and Gradle are the two main build tools used in Java projects today. JHipster allows you to use either one. With Maven, you have one `pom.xml` file that's 986 lines of XML. With Gradle, you end up with several `*.gradle` files. In the 21-Points project, their Groovy code adds up to only 583 lines.

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

Gradle is a general-purpose build tool. It can build pretty much anything you care to implement in your build script. Out of the box, however, it won't build anything unless you add code to your build script to ask for that. Gradle has a Groovy-based domain-specific language (DSL) instead of the more traditional XML form of declaring the project configuration. Like Maven, Gradle has plugins that allow you to configure tasks for your project. Most plugins add some preconfigured tasks, which together do something useful. For example, Gradle's Java plugin adds tasks to your project that will compile and unit test your Java source code as well as bundle it into a JAR file.

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

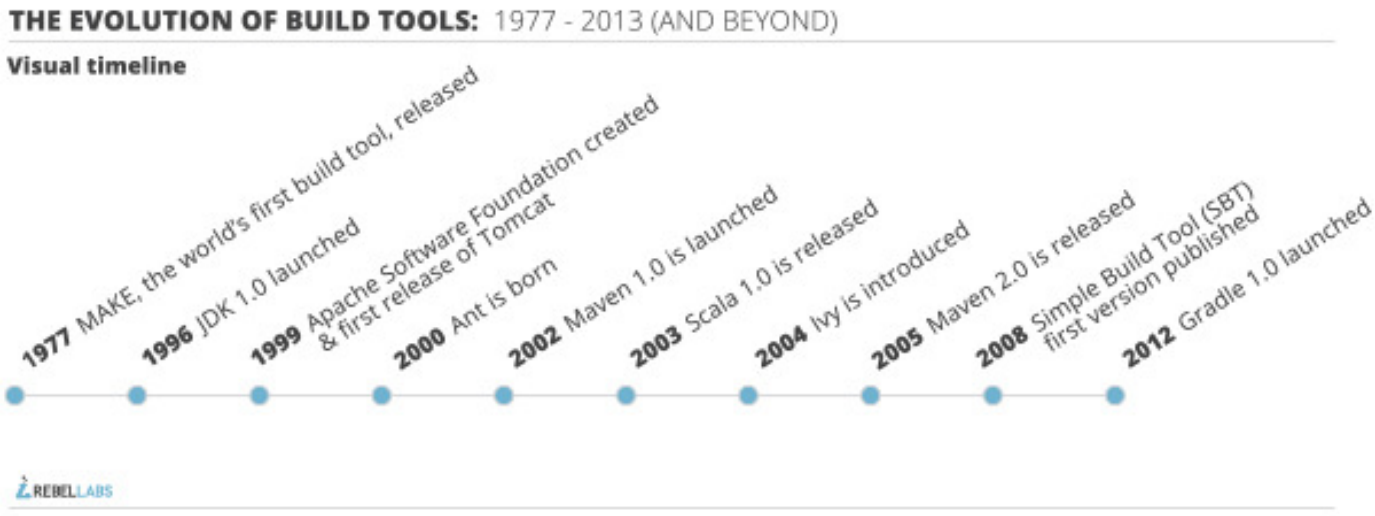

Figure 35. The Evolution of Build Tools, 1977-2013

RebelLabs advises that you should experiment with Gradle in your next project.

> If we were forced to conclude with any general recommendation, it would be to go with Gradle if you are starting a new project.
>
> — RebelLabs, Java Build Tools – Part 2: A Decision Maker's Comparison of Maven, Gradle and Ant + Ivy

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

```
<?xml version="1.0" encoding="UTF-8"?>
<extensions>
  <extension>
    <groupId>io.takari.polyglot</groupId>
    <artifactId>${artifactId}</artifactId>
    <version>0.1.19</version>
  </extension>
</extensions>
```

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

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

```
mvn io.takari.polyglot:polyglot-translate-plugin:0.1.19:translate \
 -Dinput=pom.xml -Doutput=pom.{format}
```

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

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

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

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

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

- Spring Boot Maven plugin
- Spring Boot Gradle plugin

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

IDE support: Running, debugging, and profiling

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

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

IntelliJ IDEA, which brings these same features to Java development, is a truly amazing IDE. If you're only writing JavaScript, their WebStorm IDE will likely become your best friend. Both IntelliJ products have excellent CSS support and accept plugins for many web languages/frameworks. See this tip to learn how to make IDEA auto-compile on save, like Eclipse does.

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

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

Visual Studio Code is an open-source text editor made by Microsoft. It's become a popular editor for TypeScript and has plugins for Java development.

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

To learn about profiling a Java application, I recommend you watch Nitsan Wakart's "Java Profiling from the Ground Up!" To learn more about memory and JavaScript applications, I recommend Addy Osmani's "JavaScript Memory Management Masterclass".

Security

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

The most basic Spring Security Java configuration creates a servlet `Filter`, which is responsible for all the security (protecting URLs, validating credentials, redirecting to login, etc.). This involves several lines of code, but half of them are class imports.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.*;
import org.springframework.security.config.annotation.authentication.builders.*;
import org.springframework.security.config.annotation.web.configuration.*;

@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user").password("password").roles("USER");
    }
}
```

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

- It requires authentication to every URL in your application.
- It generates a login form for you.
- It allows user:password to authenticate with form-based authentication.
- It allows the user to logout.
- It prevents CSRF attacks.
- It protects against session fixation.
- It includes security-header integration with:
  - HTTP Strict Transport Security for secure requests,
  - X-Content-Type-Options integration,
  - cache control,
  - X-XSS-Protection integration, and
  - X-Frame-Options integration to help prevent clickjacking.
- It integrates with HttpServletRequest API methods of: `getRemoteUser()`, `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:

- **HTTP Session Authentication** — Uses the HTTP session, so it is a stateful mechanism. Recommended for small applications.
- **OAuth 2.0 Authentication** — A stateless security mechanism. You might prefer it if you want to scale your application across several machines.

- **JWT authentication** — Like OAuth 2.0, 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.
- **Social Login** - Adds support for logging in with social network credentials. Supports Google, Facebook, and Twitter.

> **OAuth 2.0**
>
> OAuth 2.0 is the next version of the OAuth protocol (originally created in 2006). OAuth 2.0 focuses on simplifying client development while supporting web applications, desktop applications, mobile phones, and living-room devices. If you'd like to learn about how OAuth works, see What the Heck is OAuth?

In addition to authentication choices, JHipster offers security improvements: improved "remember me" (unique tokens stored in database), cookie-theft protection, and CSRF protection.

By default, JHipster comes with four different users:

- **system** — Used by audit logs when something is done automatically.
- **anonymousUser** — Anonymous users when they do an action.
- **user** — A normal user with "ROLE_USER" authorization; the default password is "user".
- **admin** — An admin user with "ROLE_USER" and "ROLE_ADMIN" authorizations; the default password is "admin".

For security reasons, you should change the default passwords in `src/main/resources/config/liquibase/users.csv` or through the User Management feature when deployed.

JPA versus MongoDB versus Cassandra

A traditional relational-database management system (RDBMS) provides a number of properties that guarantee its transactions are processed reliably: ACID, for atomicity, consistency, isolation, and durability. Databases like MySQL and PostgreSQL provide RDBMS support and have done wonders to reduce the costs of databases. JHipster supports vendors like Oracle and Microsoft as well. If you'd like to use a traditional database, select SQL when creating your JHipster project.

JHipster's Using Oracle guide explains how'll you need an Oracle account to download its proprietary JDBC driver.

NoSQL databases have helped many web-scale companies achieve high scalability through eventual consistency: because a NoSQL database is often distributed across several machines, with some latency, it guarantees only that all instances will eventually be consistent. Eventually consistent

services are often called BASE (basically available, soft state, eventual consistency) services in contrast to traditional ACID properties.

When you create a new JHipster project, you'll be prompted with the following.

```
? (6/16) Which *type* of database would you like to use? (Use arrow keys)
   SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
  MongoDB
  Cassandra
```

If you're familiar with RDBMS databases, I recommend you use PostgreSQL or MySQL for both development and production. PostgreSQL has great support on Heroku and MySQL has great support on AWS. JHipster's AWS sub-generator has a limitation of only working with MySQL.

If your idea is the next Facebook, you might want to consider a NoSQL database that's more concerned with performance than third normal form.

> NoSQL encompasses a wide variety of different database technologies that were developed in response to a rise in the volume of data stored about users, objects, and products, the frequency in which this data is accessed, and performance and processing needs. Relational databases, on the other hand, were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the cheap storage and processing power available today.
>
> — MongoDB, NOSQL Database Explained

MongoDB was founded in 2007 by the folks behind DoubleClick, ShopWiki, and Gilt Groupe. It uses the Apache and GNU-APGL licenses on GitHub. Its many large customers include Adobe, eBay, and eHarmony.

Cassandra is "a distributed storage system for managing structured data that is designed to scale to a very large size across many commodity servers, with no single point of failure" (from "Cassandra – A structured storage system on a P2P Network" on the Facebook Engineering blog). It was initially developed at Facebook to power its Inbox Search feature. Its creators, Avinash Lakshman (one of the creators of Amazon DynamoDB) and Prashant Malik, released it as an open-source project in July 2008. In March 2009, it became an Apache Incubator project, and graduated to a top-level project in February 2010.

In addition to Facebook, Cassandra helps a number of other companies achieve web scale. It has some impressive numbers about scalability on its homepage.

> One of the largest production deployments is Apple's, with over 75,000 nodes storing over 10 PB of data. Other large Cassandra installations include Netflix (2,500 nodes, 420 TB, over 1 trillion requests per day), Chinese search engine Easou (270 nodes, 300 TB, over 800 million requests per day), and eBay (over 100 nodes, 250 TB).
>
> — Cassandra, Project Homepage

JHipster's data support lets you dream big!

NoSQL with JHipster

When MongoDB is selected:

- JHipster will use Spring Data MongoDB, similar to Spring Data JPA.
- JHipster will use Mongobee instead of Liquibase to manage database migrations.
- The entity sub-generator will not ask you about relationships. You can't have relationships with a NoSQL database.

Cassandra does not support OAuth 2.0 authentication and it requires you to run its migration tool manually, or in a Docker container.

Liquibase

Liquibase is "source control for your database". It's an open-source (Apache 2.0) project that allows you to manipulate your database as part of a build or runtime process. It allows you to diff your entities against your database tables and create migration scripts. It even allows you to provide comma-delimited default data! For example, default users are loaded from `src/main/resources/config/liquibase/users.csv`.

This file is loaded by Liquibase when it creates the database schema.

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

```
<loadData encoding="UTF-8"
          file="config/liquibase/users.csv"
          separator=";"
          tableName="jhi_user">
    <column name="activated" type="boolean"/>
    <column name="created_date" type="timestamp"/>
</loadData>
<dropDefaultValue tableName="jhi_user" columnName="created_date" columnDataType="
datetime"/>
```

Liquibase supports most major databases. If you use MySQL or PostgreSQL, you can use `mvn liquibase:diff` (or `./gradlew liquibaseDiffChangelog`) to automatically generate a changelog.

JHipster's development guide recommends the following workflow:

1. Modify your JPA entity (add a field, a relationship, etc.).
2. Run `mvn compile liquibase:diff`.
3. A new changelog is created in your `src/main/resources/config/liquibase/changelog` directory.
4. Review this changelog and add it to your `src/main/resources/config/liquibase/master.xml` file, so it is applied the next time you run your application.

If you use Gradle, you can use the same workflow by confirming database settings in `liquibase.gradle` and running `./gradlew liquibaseDiffChangelog`.

Elasticsearch

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

When using the (default) "dev" profile, the in-memory Elasticsearch instance will store files in the `target` folder.

When I deployed 21-Points to Heroku, my app failed to start because it expected to find Elasticsearch nodes listening on `localhost:9200`. To fix this, I changed my production configuration.

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

```
data:
    elasticsearch:
        cluster-name:
        cluster-nodes:
        properties:
            path:
              logs: /tmp/elasticsearch/log
              data: /tmp/elasticsearch/data
```

Elasticsearch is used by a number of well-known companies: Facebook, GitHub, and Uber among others. The project is backed by Elastic, which provides an ecosystem of projects around Elasticsearch. Some examples are:

- Elasticsearch as a Service — "Hosted and managed Elasticsearch".
- Logstash — "Process any data, from any source".
- Kibana — "Explore and visualize your data".

The ELK (Elasticsearch, Logstash, and Kibana) stack is all open-source projects sponsored by Elastic. It's a powerful solution for monitoring your applications and seeing how they're being used.

Deployment

A JHipster application can be deployed wherever a Java program can be run. Spring Boot uses a `public static void main` entry point that launches an embedded web server for you. Spring Boot applications are embedded in a "fat JAR", which includes all necessary dependencies like, for example, the web server and start/stop scripts. You can give anybody this `.jar` and they can easily run your app: no build tool required, no setup, no web-server configuration, etc. It's just `java -jar killerapp.jar`.

Josh Long's "Deploying Spring Boot Applications" is an excellent resource for learning how to customize your application archive. It shows how to change your application to a traditional WAR: extend `SpringBootServletInitializer`, change packaging to `war`, and set `spring-boot-starter-tomcat` as a provided dependency.

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

```
mvn -Pprod spring-boot:run
```

With Gradle, it's:

```
gradlew -Pprod bootRun
```

The "prod" profile will trigger a `webpack:prod`, which optimizes your static resources. It will combine your JavaScript and CSS files, minify them, and get them production ready. It also updates your HTML (in your `(build|target)/www` directory) to have references to your versioned, combined, and minified files.

A JHipster application can be deployed to your own JVM, Cloud Foundry, Heroku, Kubernetes, and AWS.

I've deployed JHipster applications to both Heroku and Cloud Foundry. With Heroku, you might have to ask to double the timeout (from 60 to 120 seconds) to get your application started. Heroku support is usually quick to respond and can make it happen in a matter of minutes. In 2015, the JHipster team created a non-blocking Liquibase bean and cut startup time by 40%.

Summary

The Spring Framework has one of the best track records for hipness in Javaland. It's remained backwards compatible between many releases and has lived as an open-source project for more than 12 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 in Java (and Groovy) and deploy them to the cloud.

You've seen some of the cool features of Spring Boot and the build tools you can use to package and run a JHipster application. I've described the power of Spring Security and showed you its many features, which you can enable with only a few lines of code. JHipster supports both relational databases and NoSQL databases, which allows you to choose how you want your data stored. You can choose JPA, MongoDB, or Cassandra when creating a new application.

Liquibase will create your database schema for you and help you update your database when the need arises. It provides an easy-to-use workflow to adding new properties to your JHipster-generated entities using its diff feature.

You can add rich search capabilities to your JHipster app with Elasticsearch. This is one of the most popular Java projects on GitHub and there's a reason for that: it works really well.

JHipster applications are Spring Boot applications, so you can deploy them wherever Java can be run. You can deploy them in a traditional Java EE (or servlet) container or you can deploy them in the cloud. The sky's the limit!

Action!

I hope you've enjoyed learning how JHipster can help you develop hip web applications! It's a nifty project, with an easy-to-use entity generator, a pretty UI and many Spring Boot best-practice patterns. The project team follows five simple policies, paraphrased here:

1. The development team votes on policies.
2. JHipster uses technologies with their default configurations as much as possible.
3. Only add options when there is sufficient added value in the generated code.
4. For the Java code, follow the default IntelliJ IDEA formatting and coding guidelines.
5. Use strict versions for third-party libraries.

These policies help the project maintain its sharp edge and streamline its development process. If you have features you'd like to add or if you'd like to refine existing features, please follow the project and help with its development and support. We're always looking for help!

Now that you've learned how to use Angular, Bootstrap, and Spring Boot with JHipster, go forth and develop great applications!

Additional reading

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

The definitive book about Spring's JDBC, JPA, and NoSQL support is *Spring Data: Modern Data Access for Enterprise Java* by Mark Pollack *et al.* (O'Reilly Media, October 2012).

Learn how to use Spring Boot to build apps faster than ever before with *Learning Spring Boot* by Greg L. Turnquist (Packt Publishing, November 2014).

An in-depth and up-to-date book on Angular is *ng-book: The Complete Guide to Angular 4* by by Nathan Murray, Felipe Coury, Ari Lerner, and Carlos Taborda) (Fullstack.io, April 2017).

Learn how to develop cloud-ready JVM applications and microservices across a variety of environments with *Cloud Native Java: Designing Resilient Systems with Spring Boot, Spring Cloud, and Cloud Foundry* by Josh Long and Kenny Bastani (O'Reilly Media, August 2017).

About the author

Matt Raible is a Java Hipster. He grew up in the backwoods of Montana with no electricity or running water. He walked a mile and a half to the bus stop on school days. His mom and sister often led the early-morning hikes, but his BMX skills overcame this handicap later in life.

He started writing HTML, CSS, and JavaScript in the early '90s and got into Java in the late '90s. He loves the Volkswagen Bus, like no one should love anything. He has a passion for skiing, mountain biking, VWs, and good beer. Matt is married to an awesome woman and amazing photographer, Trish McGinity. They love skiing, rafting, and camping with their fun-loving kids, Abbie and Jack.

Matt's blog is at http://raibledesigns.com. You can also find him on Twitter at @mraible. Matt drives a 1966 Deluxe Samba and a 1990 Vanagon Syncro.