

UPDATED FOR JHIPSTER 3.X



# THE JHIPSTER MINI-BOOK

Matt Raible

**InfoQ**  
ENTERPRISE SOFTWARE  
DEVELOPMENT SERIES

# **THE JHIPSTER MINI-BOOK**

**Matt Raible**

## **The JHipster Mini-Book**

© 2016 Matt Raible. All rights reserved. Version 2.0.

Published by C4Media, publisher of InfoQ.com.

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

Production Editor: Ana Ciobotaru

Copy Editor: Lawrence Nyveen

Cover and Interior Design: Dragos Balasoiu

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

# Table of Contents

Dedication .....	1
Acknowledgements .....	2
Preface .....	3
What is in an InfoQ mini-book? .....	3
Who this book is for .....	3
What you need for this book .....	4
Conventions .....	4
Reader feedback .....	5
Introduction .....	6
Building an app with JHipster .....	7
Creating the application .....	9
Building the UI and business logic .....	13
Application improvements .....	22
Deploying to Heroku .....	51
Monitoring and analytics .....	58
Continuous integration and deployment .....	59
Source code .....	64
Upgrading 21-Points Health .....	65
Summary .....	66
JHipster's UI components .....	67
AngularJS .....	68
Bootstrap .....	80
Internationalization (i18n) .....	88
Sass .....	88
Gulp .....	90
WebSockets .....	94
Browsersync .....	98
Summary .....	99
JHipster's API building blocks .....	100
Spring Boot .....	101
Maven versus Gradle .....	112
IDE support: Running, debugging, and profiling .....	114
Security .....	115
JPA versus MongoDB versus Cassandra .....	117
Liquibase .....	119
Elasticsearch .....	120

Deployment .....	121
Summary .....	122
Action! .....	123
Additional reading .....	123
About the author .....	124

# 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 for creating JHipster and turning it into a widely used, 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've used AngularJS for several years as well. For the first project I used Angular 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 AngularJS, 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 v4.5.0 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/main/webapp/form.html*

```
<form ng-submit="search()">
  <input type="search" name="search" ng-model="term">
  <button>Search</button>
</form>
```

*src/main/java/demo/DemoApplication.java*

```
@RestController
class BlogController {

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

  @Autowired
  BlogRepository repository;
}
```

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

```
.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 `SearchController`, simply add it as a parameter to the controller's argument list.
- ② `$scope.search()` is a function that's called from the HTML's `<form>`, wired up using the `ng-submit` directive.
- ③ `$scope.term` is a variable that's wired to `<input>` using the `ng-model` directive.



Tips are shown using callouts like this.



Warnings are shown using callouts like this.

## Sidebar

Additional information about a certain topic may be displayed using 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](mailto:feedback@infoq.com), send a tweet to [@jhipster\\_book](#), or post a question on Stack Overflow using the “jhipster” tag.

If there is a topic in which you have expertise and you are interested in either writing or contributing to a book, please take a look at our mini-book guidelines at <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, AngularJS, 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 110 releases! It is an open-source, Apache 2.0-licensed project on GitHub. It has a core team of 16 developers and over 250 contributors. You can find its homepage at <http://jhipster.github.io>. Its [Open HUB profile](#) shows it’s mostly written in JavaScript (50%) and Java (35%).

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 webapp:

- the scaffolding tool (`yo`),
- the build tool (Grunt, Gulp, etc.), and
- the package manager (Bower, npm, 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 1996 VW Bus I've been working on for over 10 years. 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](#), then ran the following commands to install [Yeoman](#) and [Bower](#).

```
npm install -g yo  
npm install -g bower
```

I also installed Gulp and JHipster.

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

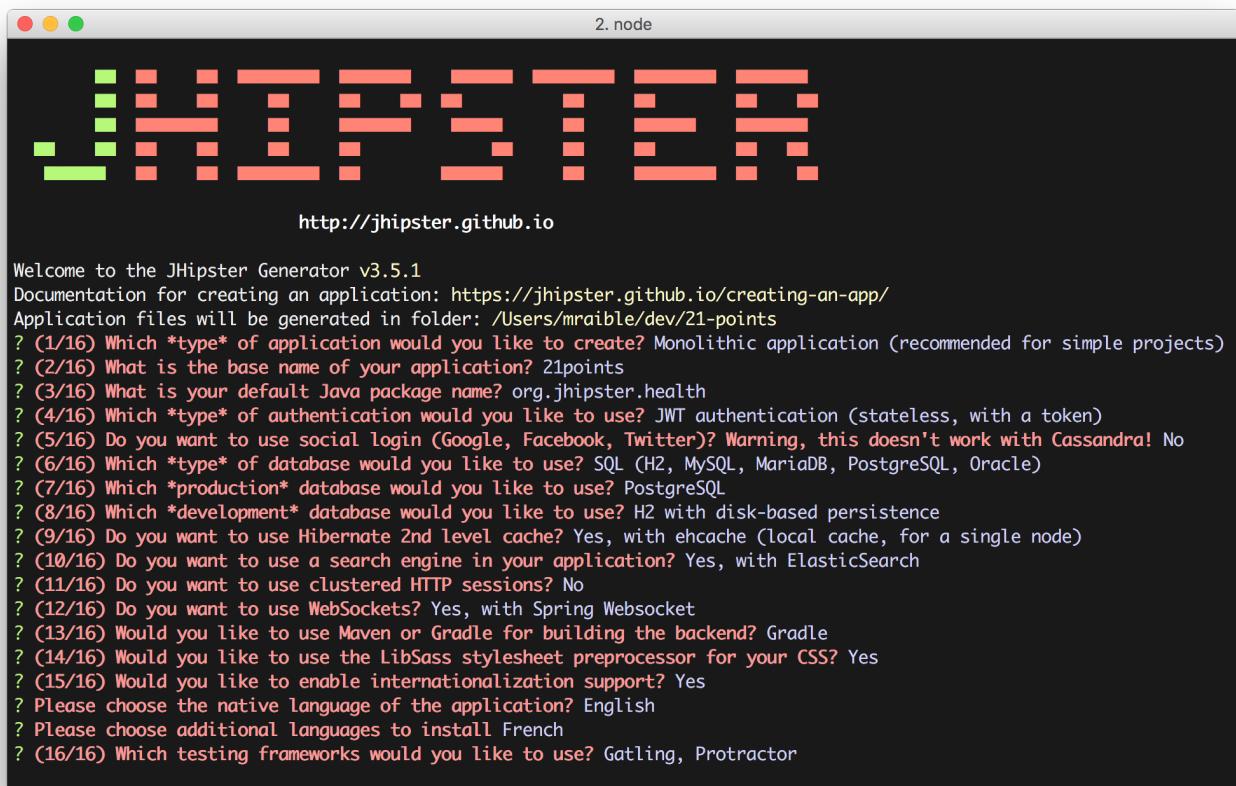


If you need to install Java, Maven or Git, please see [JHipster's local installation documentation](#).

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

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 `npm install` followed by `bower install`. To prove everything was good to go, I ran the Java unit tests using `./gradlew test`.

```
BUILD SUCCESSFUL
```

```
Total time: 36.91 secs
```

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

```
$ gulp itest
[12:31:23] Using gulpfile ~/dev/21-points/gulpfile.js
[12:31:23] Starting 'protractor'...
[12:31:24] W/configParser - pattern ./e2e/entities/*.js did not match any files.
[12:31:24] I/direct - Using FirefoxDriver directly...
[12:31:24] I/launcher - Running 1 instances of WebDriver
Started
.....
10 specs, 0 failures
Finished in 29.433 seconds
[12:31:56] I/launcher - 0 instance(s) of WebDriver still running
[12:31:56] I/launcher - firefox #01 passed
[12:31:56] Finished 'protractor' after 33 s
[12:31:56] Starting 'itest'...
[12:31:56] Finished 'itest' after 15 µs
```

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`. You can see that PostgreSQL didn't like that my database name started with a number.

```
'/Applications/Postgres.app/Contents/Versions/9.5/bin'/psql -p5432
[mraible:~] $ '/Applications/Postgres.app/Contents/Versions/9.5/bin'/psql -p5432
psql (9.5.3)
Type "help" for help.

mraible=# create user 21points with password '21points';
ERROR:  syntax error at or near "21"
LINE 1: create user 21points with password '';
          ^
mraible=# create user health with password 'health';
CREATE ROLE
mraible=# create database health;
CREATE DATABASE
mraible=# grant all privileges on database health to health;
GRANT
mraible=#

```

I chose the name “health” instead and updated [application-prod.yml](#) to use this name and the specified credentials.

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

```
datasource:
- url: jdbc:postgresql://localhost:5432/21points
- name:
- username: 21points
- password:
+ url: jdbc:postgresql://localhost:5432/health
+ name:
+ username: health
+ password: health
```

I confirmed I could talk to a PostgreSQL database when running with the `prod` profile.

```
$ ./gradlew -Pprod
...
-----
Application '21points' is running! Access URLs:
Local:      http://127.0.0.1:8080
External:    http://172.20.10.2:8080
-----
```

## 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](#). Because I don't like my Bower dependencies stored with my source code, I excluded them by adding the following to [.gitignore](#).

```
/src/main/webapp/bower_components/**
```

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 -A

$ git commit -m "Initial checkin of 21-points application"
[master (root-commit) e9880d7] Initial checkin of 21-points application
 364 files changed, 24719 insertions(+)
 ...

$ git push origin master
Counting objects: 494, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (463/463), done.
Writing objects: 100% (494/494), 470.33 KiB | 0 bytes/s, done.
Total 494 (delta 64), reused 0 (delta 0)
To git@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. To install it, I executed the following commands.

```
npm install -g generator-jhipster-bootstrap-material-design
yo jhipster-bootstrap-material-design
```

The second command yielded a number of questions about overwriting files, to which I agreed.

```
$ yo jhipster-bootstrap-material-design

   _-----_
  |       |
  |  --(o)--|  | Welcome to the JHipster   |
  |         |  | Bootstrap Material      |
  |  ( _U_ ) |  | design generator! v3.5.1 |
  | /__A__\  |  '-----'
  |    ~   |
  '-----'` 

? Do you want to install Bootstrap Material design? Yes
Composing JHipster configuration with module bootstrap-material-design
Reading the JHipster project configuration for your module
conflict bower.json
? Overwrite bower.json? overwrite
  force bower.json
  create src/main/webapp/app/blocks/config/bootstrap-material.config.js
  conflict src/main/webapp/scss/main.scss
? Overwrite src/main/webapp/scss/main.scss? overwrite
  force src/main/webapp/scss/main.scss
  conflict src/main/webapp/content/css/main.css
? Overwrite src/main/webapp/content/css/main.css? overwrite
  force src/main/webapp/content/css/main.css
```

In the terminal log, I noticed a message:

You are using SASS! Please make sure to check the documentation on using SASS:  
<https://github.com/moifort/generator-jhipster-bootstrap-material-design>

I visited the [module's documentation](#) and learned about the changes I needed to make in [index.html](#).

## Using Sass

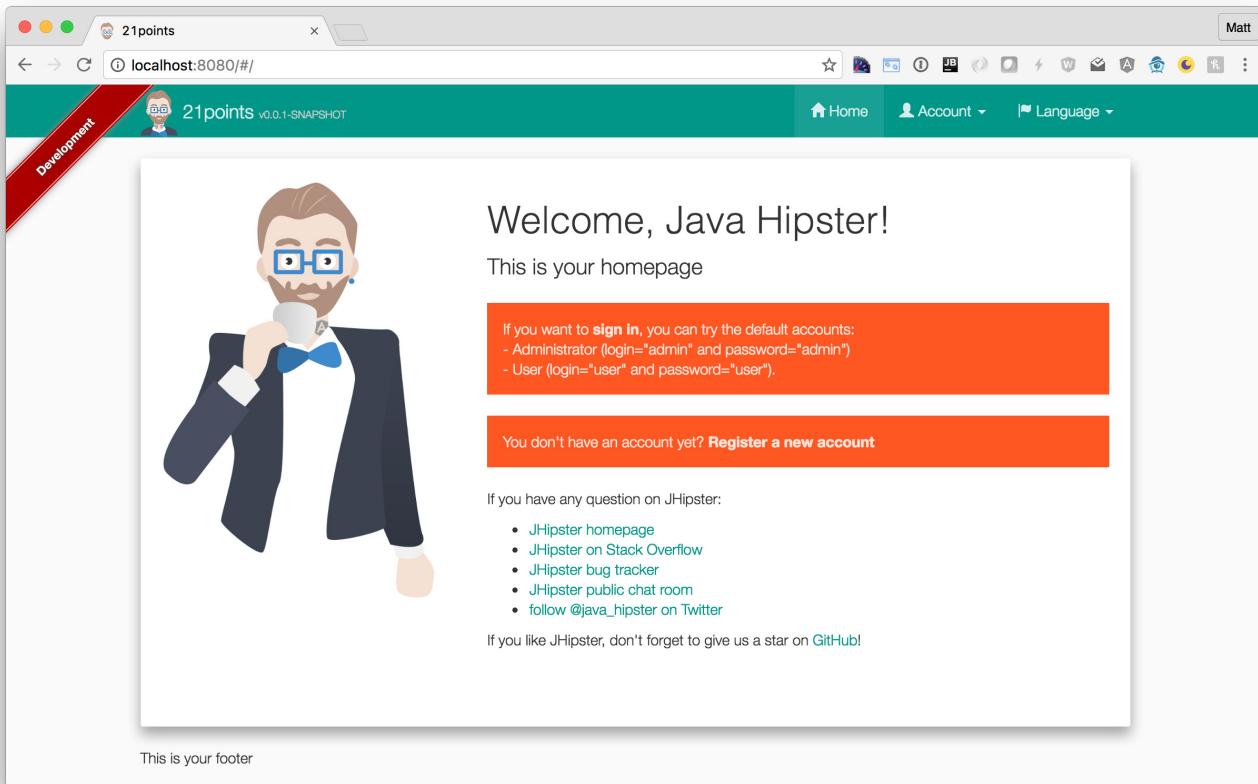
When you use Sass, you need to modify your `index.html` a little bit to make sure your material-design styles are not overwritten by the default bootstrap styles:

```
<!-- build:css content/css/vendor.css -->
<link rel="stylesheet" href="content/css/vendor.css">
<!-- bower:css -->
<link rel="stylesheet" href="bower_components/angular-loading-bar/build/loading-
bar.css">
<link rel="stylesheet" href="bower_components/bootstrap-material-
design/dist/css/bootstrap-material-design.css">
<link rel="stylesheet" href="bower_components/bootstrap-material-
design/dist/css/ripples.css">
<!-- endinject -->
<!-- endbuild -->
```

You should make sure that the `vendor.css` is before all material-design stylesheets in your `index.html`!

After doing this, I observed the changes made by installing this module using `git diff`.

Finally, I ran `./gradlew` and confirmed that the new theme was being used.



*Figure 2. Material design for Bootstrap theme*

Before creating the entities and associated database tables for this application, I decided to upgrade JHipster to the latest release. You can see that I created this application with JHipster 3.5.1. The latest release as of writing is 3.7.0, so I updated my version with the following command.

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

This installs the latest version of JHipster, but does nothing to upgrade my project. I had to run the following command to update the project.

```
yo jhipster
```

This notified me that it was deleting a number of files, and that there were some conflicts in my files.



If you don't see conflicts when upgrading, it's possible that you didn't install JHipster on the machine you're using. This happened to me when I switched machines. Check `package.json` to ensure it has the new version number. If it does not, run `npm install generator-jhipster@latest --save`.

This is an existing project, using the configuration from your `.yo-rc.json` file to re-generate the project...

```
Installing languages: en, fr
conflict bower.json
? Overwrite bower.json? (Ynaxdh)
```

I answered “Y” to all the conflict questions. Because I had the files in source control, I was able to diff the changes after they were made and decide if I wanted them or not. When I tried to start the app, I did experience a `ConflictingBeanDefinitionException` exception for `userMapperImpl`. I was able to solve this by deleting `org/jhipster/health/web/rest/mapper/UserMapper.java`.

I ran `gulp` to verify that everything looked good, and saw the following error in my console.

```
bootstrap-material.config.js:8 Uncaught ReferenceError: compileServiceConfig is not
defined
```

I opened `bootstrap-material.config.js` to look for the problem.

```
angular
  .module('21PointsApp')
  .config(bootstrapMaterialDesignConfig);

  compileServiceConfig.$inject = [];

  function bootstrapMaterialDesignConfig() {
    $material.init();
  }
```

I changed `compileServiceConfig.$inject = [];` to `bootstrapMaterialDesignConfig.$inject = []` and it cleaned up my console. I then [logged an issue](#) for this bug.

After making this change, I committed my updated project to Git. Then I ran `./gradlew test` to make sure all unit tests passed. I followed this up with `./gradlew` in one terminal window and `gulp itest` (in the same directory) in another.

```
10 specs, 0 failures
Finished in 33.065 seconds
[00:35:55] I/launcher - 0 instance(s) of WebDriver still running
[00:35:55] I/launcher - firefox #01 passed
[00:35:55] Finished 'protractor' after 37 s
[00:35:55] Starting 'itest'...
[00:35:55] Finished 'itest' after 28 µs
[mraible:~/dev/21-points] master(+321/-400) 40s $
```

I opened the application in a browser and noticed there were some rounded corners and the primary buttons looked funny. This happened because the upgrade overwrote [index.html](#) and I had to re-apply the [Using Sass](#) fix.



After integrating the material-design theme, 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 AngularJS 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 JavaScript 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 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.

The UI mockup consists of five screens:

- 21 Point Health:** Shows a progress bar for "Points this week" from 12 to 21. Below it are sections for "Weight" (with a "Graph" button) and "Blood Pressure" (with a "Graph" button). A "View History" link is at the bottom.
- Add Weight:** A form with fields for "Date / Time" and "Pounds / Kilograms", and buttons for "Save" and "Cancel".
- Add Blood Pressure:** A form with fields for "Date / Time", "Systolic", and "Diastolic", and buttons for "Save" and "Cancel".
- Enter Points:** A form with a "Date" input, checkboxes for "Exercise", "Meals", and "Alcohol", a "Notes" text area, and buttons for "Save" and "Cancel".
- Settings:** A form with "Points per Week Goal" (Number of points), "Weight Units" (Pounds dropdown), and buttons for "Save" and "Cancel".

Figure 3. UI mockup

After figuring out how I wanted the UI to look, I started to think about the data model. I quickly decided I didn't need to track high-level goals (e.g. lose five pounds in Q4 2016). I was more concerned with tracking weekly goals and 21-Points Health is all about how many points you get in a week. I created the following diagram as my data model.

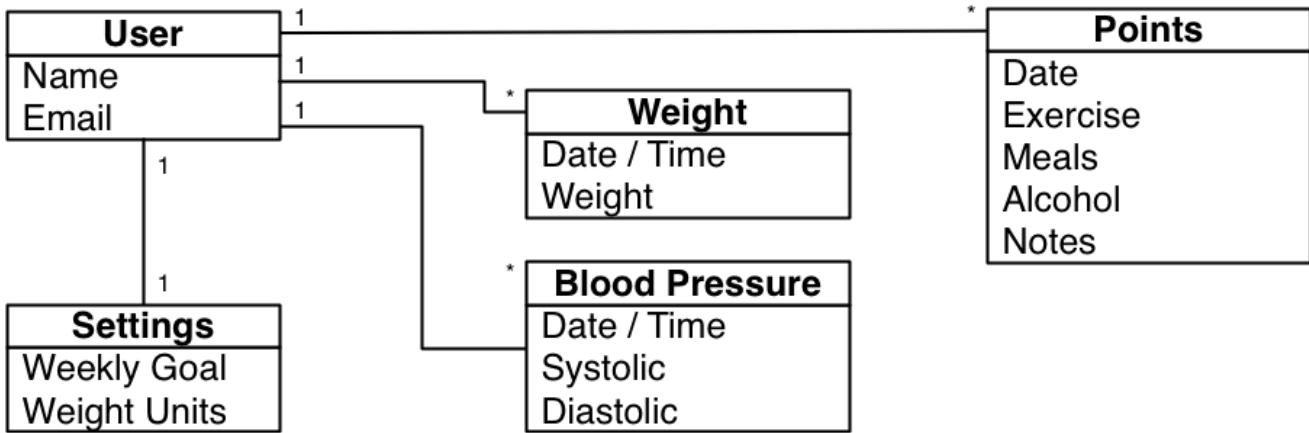


Figure 4. 21-Points Health entity diagram

I ran `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 infinite scroll
```

Everything is configured, generating the entity...

```
create .jhipster/Points.json
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/repository/search/PointsSearchRepository.java
create src/main/java/org/jhipster/health/web/rest/PointsResource.java
create
src/main/resources/config/liquibase/changelog/20160831020048_added_entity_Points.xml
create
src/main/resources/config/liquibase/changelog/20160831020048_added_entity_constraints_Points.xml
```

```

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
  create src/main/webapp/app/entities/points/points.html
  create src/main/webapp/app/entities/points/points-detail.html
  create src/main/webapp/app/entities/points/points-dialog.html
  create src/main/webapp/app/entities/points/points-delete-dialog.html
  force src/main/webapp/app/layouts/navbar/navbar.html
  create src/main/webapp/app/entities/points/points.state.js
  create src/main/webapp/app/entities/points/points.controller.js
  create src/main/webapp/app/entities/points/points-dialog.controller.js
  create src/main/webapp/app/entities/points/points-delete-dialog.controller.js
  create src/main/webapp/app/entities/points/points-detail.controller.js
  create src/main/webapp/app/entities/points/points.service.js
  create src/main/webapp/app/entities/points/points.search.service.js
  create src/main/webapp/i18n/en/points.json
    force src/main/webapp/i18n/en/global.json
  create src/main/webapp/i18n/fr/points.json
    force src/main/webapp/i18n/fr/global.json
  create src/test/javascript/spec/app/entities/points/points-detail.controller.spec.js
  create src/test/javascript/e2e/entities/points.js
  create src/test/java/org/jhipster/health/web/rest/PointsResourceIntTest.java
  create src/test/gatling/simulations/PointsGatlingTest.scala

```

Running gulp Inject to add javascript to index

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: 17.64 secs
```

I checked in four changed files and 99 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 AngularJS's date filter (for example:  `{{bloodPressure.timestamp | date: 'short': 'UTC'}}` ),
- improved dialogs to hide ID when creating a new entity,
- 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.

ID	Date	Did you exercise?	Did you eat well?	Did you drink responsibly?	Notes	User
1	Sep 9, 2016	1	1	1	Walked dogs at noon, leftovers for lunch. Flew to San Mateo for interviews.	user <a href="#">VIEW</a> <a href="#">EDIT</a> <a href="#">DELETE</a>
2	Sep 10, 2016	1	1	0	Devoxx4Kids Denver with Venkat! Dream come true. Walked the dogs in the afternoon. Carp Slam X After Party.	user <a href="#">VIEW</a> <a href="#">EDIT</a> <a href="#">DELETE</a>
3	Sep 11, 2016	1	1	1	Bike ride to downtown. Walk through of the Raible Ranch. Moving soooon!	user <a href="#">VIEW</a> <a href="#">EDIT</a> <a href="#">DELETE</a>

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

Figure 5. Default Daily Points list

Date	Did you exercise?	Did you eat well?	Did you drink responsibly?	Notes	User
Sep 9, 2016	✓	✓	✓	Walked dogs at noon, lefto...	user <a href="#">VIEW</a> <a href="#">EDIT</a> <a href="#">DELETE</a>
Sep 10, 2016	✓	✓	✗	Devoxx4Kids Denver with V...	user <a href="#">VIEW</a> <a href="#">EDIT</a> <a href="#">DELETE</a>
Sep 11, 2016	✓	✓	✓	Bike ride to downtown. Wal...	user <a href="#">VIEW</a> <a href="#">EDIT</a> <a href="#">DELETE</a>
Sep 12, 2016	✓	✓	✗	Got the offer from Stormpa...	user <a href="#">VIEW</a> <a href="#">EDIT</a> <a href="#">DELETE</a>
Sep 13, 2016	✓	✓	✓	Took the boys to Mt Falcon...	user <a href="#">VIEW</a> <a href="#">EDIT</a> <a href="#">DELETE</a>
Sep 14, 2016	✓	✓	✗	Double Workout day. Took ...	user <a href="#">VIEW</a> <a href="#">EDIT</a> <a href="#">DELETE</a>

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

Figure 6. Default Daily Points list after UI improvements

I refactored the HTML at the top of `points.html` to put the title, search, and add buttons on the same row. I also removed the button text in favor of using [UI Bootstrap's tooltip directive](#). The `translate` filter you see in the button titles is provided by [angular-translate](#). Both UI Bootstrap and angular-translate are included in JHipster by default.

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

```

<div class="row">
  <div class="col-sm-7">
    <h2 translate="21PointsApp.points.home.title">Points</h2>
  </div>
  <div class="col-sm-5 text-right p-n">
    <form name="searchForm" class="form-inline">
      <div class="form-group m-n">
        <input type="search" class="form-control" ng-model="vm.searchQuery" id="searchQuery"
               placeholder="{{'entity.action.search' | translate}}">
      </div>
      <button class="btn btn-info btn-sm" ng-click="vm.clear()" ng-if="vm.currentSearch">
        <span class="glyphicon glyphicon-trash"></span>
      </button>
      <button class="btn btn-info btn-sm btn-raised" ng-click="vm.search(vm.searchQuery)"
             uib-tooltip="{{'entity.action.search' | translate}}">
        <i class="glyphicon glyphicon-search"></i>
      </button>
      <button class="btn btn-primary btn-sm btn-raised" ui-sref="points.new"
             uib-tooltip="{{'entity.action.new' | translate}}">
        <span class="glyphicon glyphicon-plus"></span>
      </button>
    </form>
  </div>
</div>

```

Changing the numbers to icons was pretty easy thanks to Angular's `ng-class` directive.

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

```
<td class="text-center">
  <i class="glyphicon"
    ng-class="{'glyphicon-ok text-success': points.exercise,
              'glyphicon-remove text-danger': !points.exercise}"></i>
</td>
<td class="text-center">
  <i class="glyphicon"
    ng-class="{'glyphicon-ok text-success': points.meals,
              'glyphicon-remove text-danger': !points.meals}"></i>
</td>
<td class="text-center">
  <i class="glyphicon"
    ng-class="{'glyphicon-ok text-success': points.alcohol,
              'glyphicon-remove text-danger': !points.alcohol}"></i>
</td>
```

Similarly, I changed the input fields to checkboxes in `points-dialog.html`. Angular's `ng-true-value` and `ng-false-value` made it easy to continue receiving/sending integers to the API.

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

```
<div class="form-group">
  <div class="checkbox">
    <label>
      <input type="checkbox" ng-model="vm.points.exercise" id="field_exercise"
             ng-true-value="1" ng-false-value="0">
      <span class="check"></span>
      <label data-translate="21PointsApp.points.exercise" for="field_exercise"
            class="control-label">Exercise</label>
    </label>
  </div>
</div>
```

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

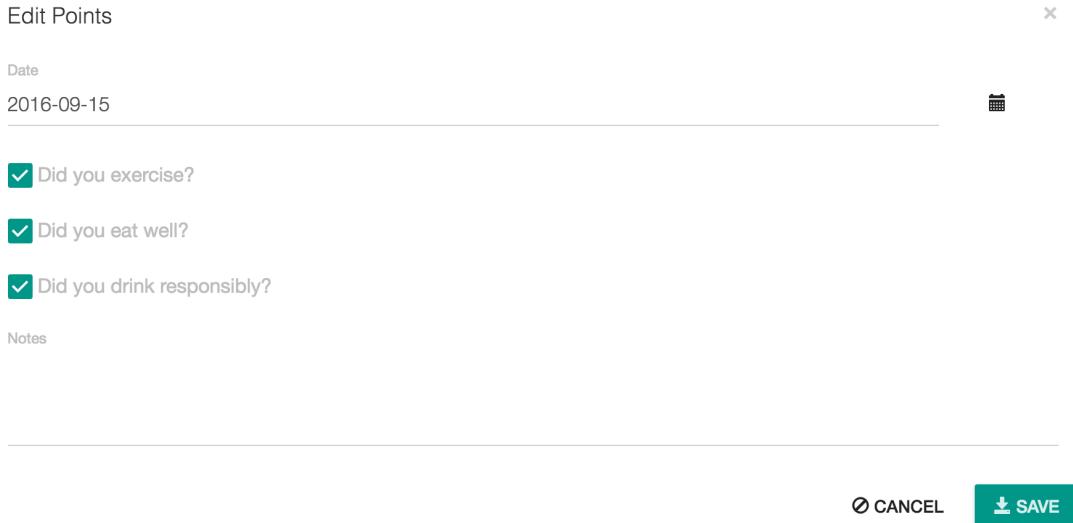


Figure 7. Add Points dialog

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 `has-authority` directive. This directive is included with JHipster, in `src/main/webapp/app/services/auth/has-authority.directive.js`. It also has a `has-any-authority` directive that allows you to pass in a comma-delimited list of roles.

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

```
<div class="form-group" has-authority="ROLE_ADMIN">
    <label class="control-label" data-translate="21PointsApp.points.user" for="field_user">User</label>
    <select class="form-control" id="field_user" name="user" ng-model="vm.points.user" ng-options="user as user.login for user in vm.users track by user.id">
        <option value=""></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`

```

@Inject
private PointsSearchRepository pointsSearchRepository;

+ @Inject
+ private UserRepository userRepository;
+
/** 
 * POST /points : Create a new points.
 *
@@ -59,6 +67,10 @@ public class PointsResource {
    if (points.getId() != null) {
        return
 ResponseEntity.badRequest().headers(HeaderUtil.createFailureAlert("points", "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);

```

`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
.*;

```

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.repository.UserRepository;
+import org.springframework.beans.factory.annotation.Autowired;
+import org.springframework.web.context.WebApplicationContext;
+import static
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessor
s.user;
+import static
org.springframework.security.test.web.servlet.setup.SecurityMockMvcConfigurers.springSecu
rity;

@@ -55,6 +63,9 @@
    @Inject
+    private UserRepository userRepository;
+
+    @Inject
        private PointsRepository pointsRepository;

    @Inject
@@ -69,6 +80,9 @@ public class PointsResourceIntTest {
    @Inject
        private EntityManager em;

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

    private Points points;

@@ -79,6 +93,7 @@
        PointsResource pointsResource = new PointsResource();
        ReflectionTestUtils.setField(pointsResource, "pointsSearchRepository",
pointsSearchRepository);
        ReflectionTestUtils.setField(pointsResource, "pointsRepository",
pointsRepository);
+        ReflectionTestUtils.setField(pointsResource, "userRepository", userRepository);
        this.restPointsMockMvc = MockMvcBuilders.standaloneSetup(pointsResource)
            .setCustomArgumentResolvers(pageableArgumentResolver)
            .setMessageConverters(jacksonMessageConverter).build();

@@ -112,9 +127,15 @@
    public void createPoints() throws Exception {
        int databaseSizeBeforeCreate = pointsRepository.findAll().size();

-        // Create the Points
+
// 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(Pageable pageable) throws
URISyntaxException {
    log.debug("REST request to get a page of Points");
    Page<Points> page;
    if (SecurityUtils.isCurrentUserInRole(AuthoritiesConstants.ADMIN)) {
        page = pointsRepository.findAllByOrderByIdDesc(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"))
        .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 found that adding new routes to `home.state.js` was the easiest way to do this. Instead of routing back to the list screen after a save succeeded, I routed the user back to the `home` state. I copied the generated `points.new` state from `points.state.js` and pasted it into `home.state.js`.

`src/main/webapp/app/home/home.state.js`

```
.state('points.add', { ①
  parent: 'home', ②
  url: 'add/points', ③
  data: {
    authorities: ['ROLE_USER']
  },
  onEnter: ['$stateParams', '$state', '$uibModal', function($stateParams, $state,
$uibModal) {
    $uibModal.open({
      templateUrl: 'app/entities/points/points-dialog.html',
      controller: 'PointsDialogController',
      controllerAs: 'vm',
      backdrop: 'static',
      size: 'lg',
      resolve: {
        entity: function () {
          return {date: null, exercise: null, meals: null, alcohol: null,
notes: null, id: null};
        }
      }
    }).result.then(function() { ④
      $state.go('home', null, { reload: true });
    }, function() {
      $state.go('home');
    });
  }]
});
```

① I changed from 'points.new' to 'points.add'.

② I changed the parent to be 'home'.

③ I changed the `url` from '/new' to 'add/points'.

- ④ I changed both result states to be 'home' instead of 'points'.

After configuring the state to add new points from the homepage, I added a button to activate the dialog.

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

```
<a ui-sref="points.add" class="btn btn-primary btn-raised">Add Points</a>
```

I fired up the application to test the button and dialog and discovered that the messages in the dialog were not translated. I fixed this by adding `$translatePartialLoader.addPart('points')` to the `home` state in `home.state.js`:

*src/main/webapp/app/home/home.state.js*

```
resolve: {
    mainTranslatePartialLoader: ['$translate', '$translatePartialLoader', function
    ($translate,$translatePartialLoader) {
        $translatePartialLoader.addPart('home');
        $translatePartialLoader.addPart('points');
        return $translate.refresh();
    }]
}
```

## 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);
}

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

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

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

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

```

① To simplify testing, I added a new constructor to `Points.java` that contained the arguments I wanted

to set. I continued this pattern for most tests I created.

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

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

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

import java.time.LocalDate;

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

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

    public Integer getPoints() {
        return points;
    }

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

    public LocalDate getWeek() {
        return week;
    }

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

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

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

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

```
List<Points> 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.
 */
@RequestMapping(value = "/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.findAllByDateBetween(startOfWeek, endOfWeek);
    // filter by current user and sum the points
    Integer numPoints = points.stream()
        .filter(p -> p.getUser().getLogin().equals(SecurityUtils.getCurrentUserLogin()))
        .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 my `Points` service.

*src/main/webapp/app/entities/points/points.service.js*

```
function Points ($resource, DateUtils) {
    var resourceUrl = 'api/points/:id';

    return $resource(resourceUrl, {}, {
        'query': { method: 'GET', isArray: true},
        'thisWeek': { method: 'GET', isArray: false, url: 'api/points-this-week'},
        ...
    });
}
```

Then I added the service to `home.controller.js` and calculated the data I wanted to display.

*src/main/webapp/app/home/home.controller.js*

```
function HomeController ($scope, Principal, LoginService, Points, $state) {
    var vm = this;

    ...

    function getAccount() {
        Principal.identity().then(function(account) {
            vm.account = account;
            vm.isAuthenticated = Principal.isAuthenticated;
        });
        Points.thisWeek(function(data) {
            vm.pointsThisWeek = data;
            vm.pointsPercentage = (data.points / 21) * 100;
        });
    }
}
```

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

src/main/webapp/app/main/main.html

```

<div class="row">
  <div class="col-md-10">
    <div class="progress progress-lg" ng-show="vm.pointsThisWeek.points"> ①
      <div class="progress-bar progress-bar-success progress-bar-striped" role="progressbar"
          aria-valuenow="{{vm.pointsThisWeek.points}}"
          aria-valuemin="0" aria-valuemax="21" style="width: {{vm.pointsPercentage}}%>
        {{vm.pointsThisWeek.points}} / Goal: 10
      </div>
    </div>
    <alert type="info" ng-hide="vm.pointsThisWeek.points">
      No points yet this week, better get moving!
    </alert>
  </div>
</div>

```

① I later realized this could be replaced with UI Bootstrap's [progressbar](#), but why fix something if it isn't broke?! 😊

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

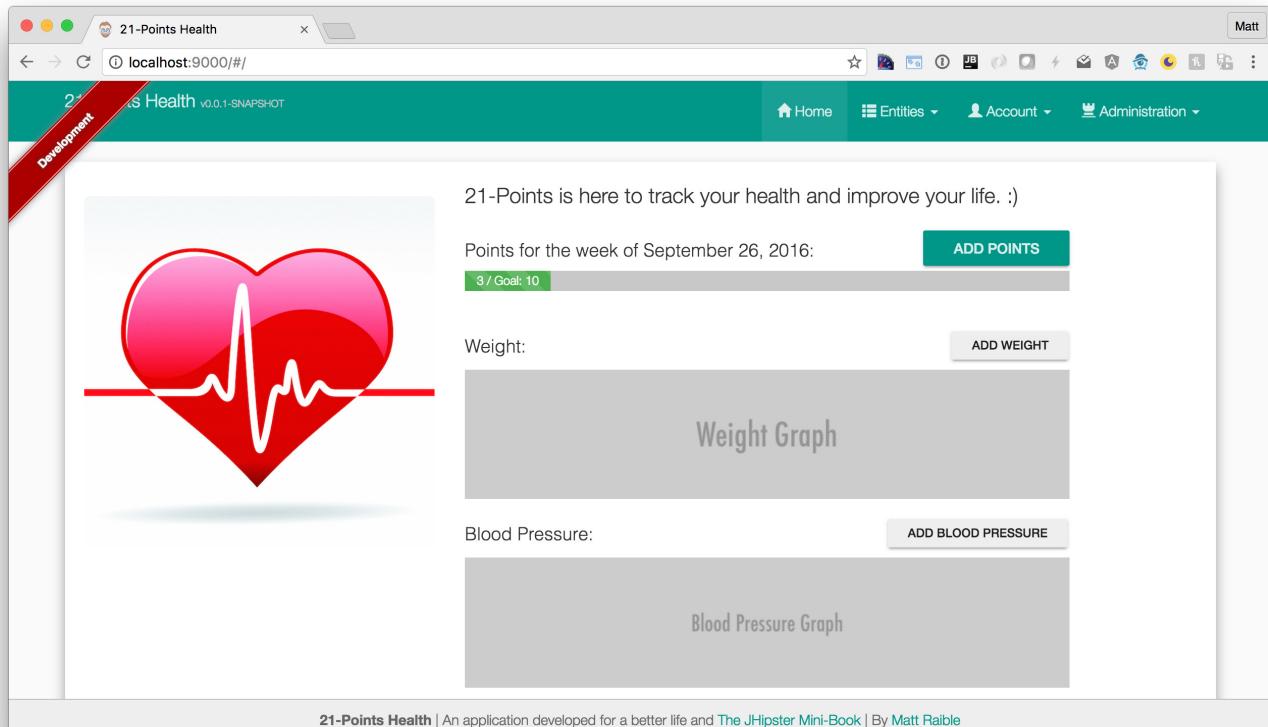


Figure 8. Progress bar for points this week

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

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

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

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

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

```
/**
 * GET /my-preferences -> get the current user's preferences.
 */
@RequestMapping(value = "/my-preferences",
    method = RequestMethod.GET,
    produces = MediaType.APPLICATION_JSON_VALUE)
@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 `Preferences` client service.

*src/main/webapp/app/entities/preferences/preferences.service.js*

```
function Preferences ($resource) {
    var resourceUrl = 'api/preferences/:id';

    return $resource(resourceUrl, {}, {
        'query': { method: 'GET', isArray: true},
        'user': { method: 'GET', isArray: false, url: '/api/my-preferences'}
        ...
    });
}
```

In `home.controller.js`, I added the `Preferences` service as a dependency and set the preferences on `vm` so the HTML template could read it.

*src/main/webapp/app/home/home.controller.js*

```
function HomeController ($scope, Principal, LoginService, Points, Preferences, $state) {
    var vm = this;
    ...

    function getAccount() {
        ...
        Preferences.user(function(data) {
            vm.preferences = data;
        });
    }
}
```

Now that a user's preferences were available, I modified `home.html` to display the user's weekly goal, as well as to color the progress bar appropriately with `ng-class`.

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

```
<div class="progress-bar progress-bar-striped" role="progressbar"
    ng-class="{'progress-bar-success': vm.pointsThisWeek.points >=
    vm.preferences.weeklyGoal,
        'progress-bar-danger': vm.pointsThisWeek.points < 10,
        'progress-bar-warning': vm.pointsThisWeek.points > 10 &&
    vm.pointsThisWeek.points < vm.preferences.weeklyGoal}"
    aria-valuenow="{{vm.pointsThisWeek.points}}"
    aria-valuemin="0" aria-valuemax="21" style="width: {{vm.pointsPercentage}}%">
    <span ng-show="vm.pointsThisWeek.points">
        {{vm.pointsThisWeek.points}} / Goal: {{vm.preferences.weeklyGoal}}
    </span>
    <span class="sr-only">{{vm.pointsPercentage}} points this week</span>
</div>
```

To finish things off, I added a link to a dialog where users could edit their preferences. I also added an appropriate state to allow editing in `home.state.js`.

### Blood pressure and weight for the last 30 days

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

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

```

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

```
function BloodPressure ($resource, DateUtils) {
    var resourceUrl = 'api/blood-pressures/:id';

    return $resource(resourceUrl, {}, {
        'query': { method: 'GET', isArray: true},
        'last30Days': { method: 'GET', isArray: false, url: 'api/bp-by-days/30'},
        ...
    });
}
```

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

I did a [bit of research](#) and decided to use [Angular-nvD3](#). I'd heard good things about [D3.js](#) and Angular-nvD3 is built on top of it. To install Angular-nvD3, I used Bower's [install](#) command.

```
bower install angular-nvd3 --save
```

Then I ran [gulp inject](#) to update [index.html](#) and [karma.conf.js](#) with references to the new files. I also updated [app.module.js](#) to add [nvd3](#) as a dependency.

*src/main/webapp/app/app.module.js*

```
angular
  .module('21PointsApp', [
    'ngStorage',
    'tmh.dynamicLocale',
    'pascalprecht.translate',
    'ngResource',
    'ngCookies',
    'ngAria',
    'ngCacheBuster',
    'ngFileUpload',
    'ui.bootstrap',
    'ui.bootstrap.datetimepicker',
    'ui.router',
    'infinite-scroll',
    // jhipster-needle-angularjs-add-module JHipster will add new module here
    'angular-loading-bar',
    'nvd3'
  ])
  .run(run);
```

I modified [home.controller.js](#) to have the [BloodPressure](#) service as a dependency and went to work building the data so Angular-nvD3 could render it. I found that charts required a bit of JSON to configure them, so I created a service to contain this configuration.

*app/components/chart/chart.service.js*

```
(function() {
  'use strict';

  angular.module('21PointsApp')
    .factory('Chart', function Chart() {
      return {
        getBpChartConfig: function() {
          return bpChartConfig;
        }
      };
    });
})()
```

```

        }
    });
});

var today = new Date();
var priorDate = new Date(). setDate(today.getDate()-30);

var bpChartConfig = {
    chart: {
        type: "lineChart",
        height: 200,
        margin: {
            top: 20,
            right: 20,
            bottom: 40,
            left: 55
        },
        x: function(d){ return d.x; },
        y: function(d){ return d.y; },
        useInteractiveGuideline: true,
        dispatch: {},
        xAxis: {
            axisLabel: "Dates",
            showMaxMin: false,
            tickFormat: function(d){
                return d3.time.format("%b %d")(new Date(d));
            }
        },
        xDomain: [priorDate, today],
        yAxis: {
            axisLabel: "",
            axisLabelDistance: 30
        },
        transitionDuration: 250
    },
    title: {
        enable: true
    }
};
})();
}
});
```

In `home.controller.js`, I grabbed the blood pressure readings from the API and morphed them into data that Angular-nvD3 could understand.

`src/main/webapp/app/home/home.controller.js`

```
BloodPressure.last30Days(function (bpReadings) {
    vm.bpReadings = bpReadings;
    if (bpReadings.readings.length) {
        vm.bpOptions = angular.copy(Chart.getBpChartConfig());
        vm.bpOptions.title.text = bpReadings.period;
        vm.bpOptions.chart.yAxis.axisLabel = "Blood Pressure";
        var systolics, diastolics, upperValues, lowerValues;
        systolics = [];
        diastolics = [];
        upperValues = [];
        lowerValues = [];
        bpReadings.readings.forEach(function (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);
        });
        vm.bpData = [
            {
                values: systolics,
                key: 'Systolic',
                color: '#673ab7'
            },
            {
                values: diastolics,
                key: 'Diastolic',
                color: '#03a9f4'
            }
        ];
        // set y scale to be 10 more than max and min
        vm.bpOptions.chart.yDomain = [Math.min.apply(Math, lowerValues) - 10, Math.max
            .apply(Math, upperValues) + 10]
    }
});
```

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

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

```
<div class="row">
  <div class="col-md-10">
    <span ng-if="vm.bpReadings.readings.length">
      <nvd3 options="vm.bpOptions" data="vm.bpData" class="with-3d-shadow with-transitions"></nvd3>
    </span>
    <span ng-if="!vm.bpReadings.readings.length">
      <div uib-alert type="info">No blood pressure readings found.</div>
    </span>
  </div>
</div>
```

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

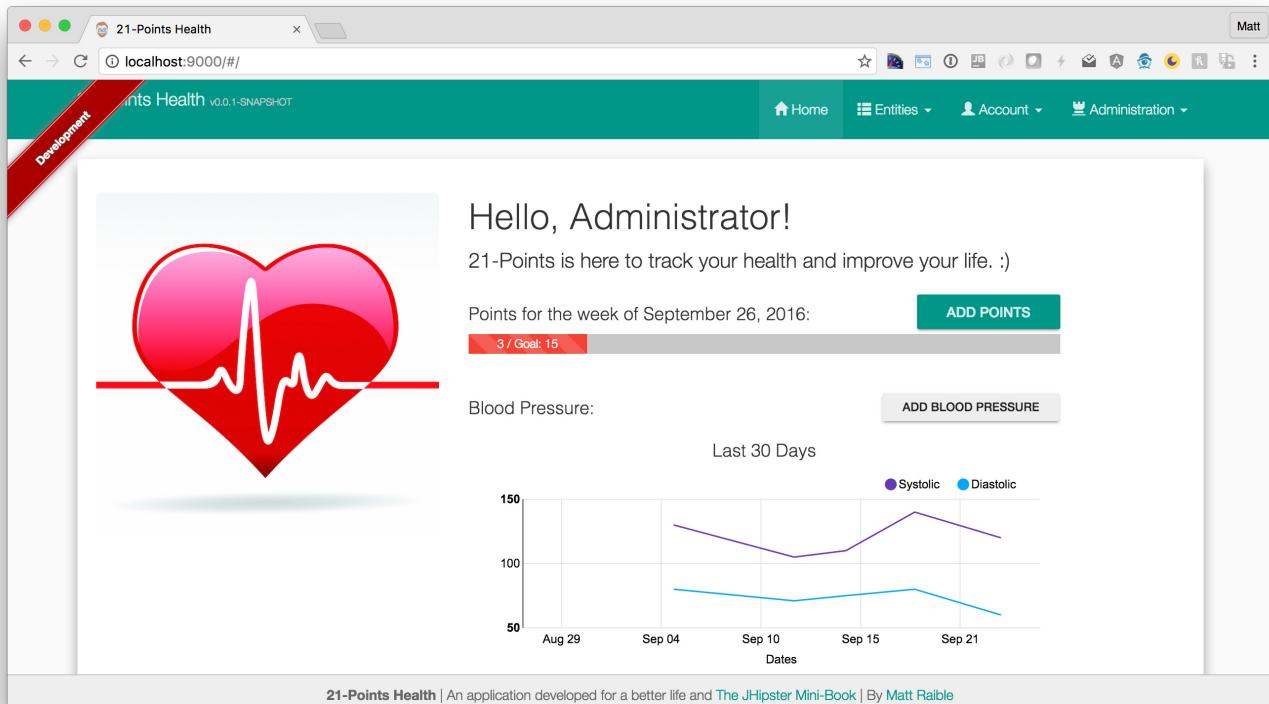
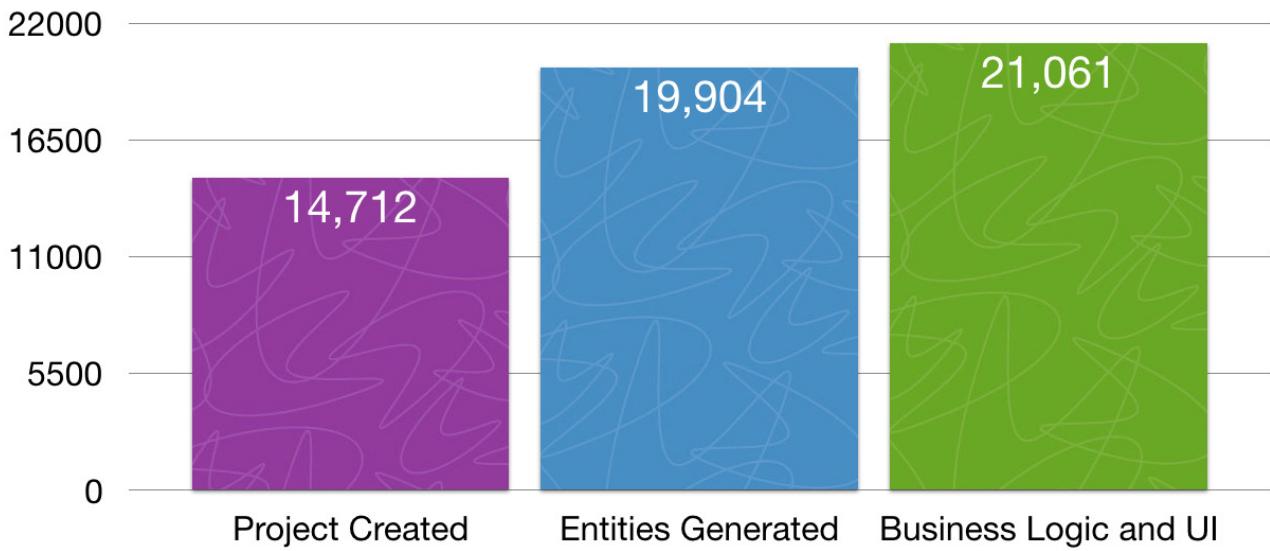


Figure 9. Chart of blood pressure during the last 30 days

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

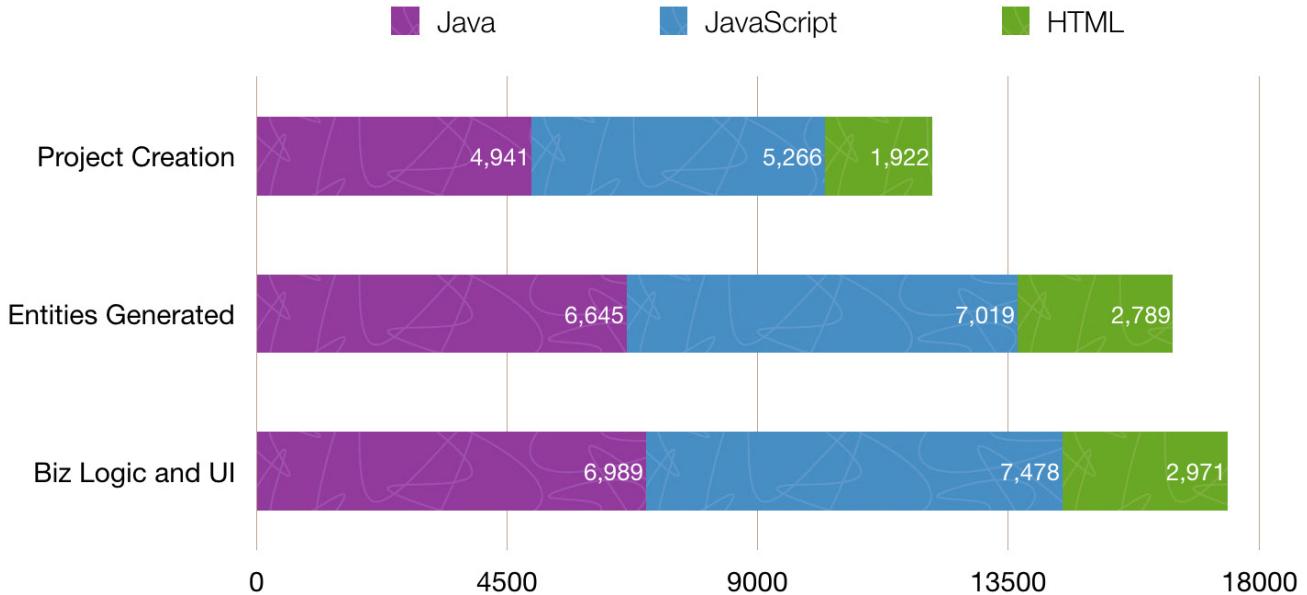
## Lines of code

After finishing the MVP (minimum viable product) of 21-Points Health, I did some quick calculations to see how many lines of code JHipster had produced. You can see from the graph below that I only had to write 1,157 lines of code. JHipster did the rest for me, generating 94.5% of the code in my project!



*Figure 10. Project lines of code*

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



*Figure 11. Project lines of code by language*

The amount of code I had to write in each language was 344 lines of Java, 459 lines of JavaScript, and 182 lines of HTML. The other 172 lines were Sass (130), JSON (25), XML (1), YAML (15), and Shell (1).

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 controllers and directives using [Jasmine](#) and I like to write integration tests with [Protractor](#).

I did not write any JavaScript tests for this project 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, 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.

```
mvn -Pprod package
```

The production profile is used to build an optimized JavaScript client. You can invoke this using Gulp by running [gulp build](#). 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.yaml](#) 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 integrating the material-design theme, meaning that I had a basically 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 --addons heroku-postgresql
{ [Error: Command failed: /bin/sh -c heroku create 21-points --addons heroku-
postgresql
Creating 21-points... !
  Name must start with a letter and can only contain lowercase letters,
  numbers, and dashes.
]
killed: false,
code: 1,
signal: null,
cmd: '/bin/sh -c heroku create 21-points --addons heroku-postgresql' }
```

You can see my first attempt failed for the same reason that creating a local PostgreSQL database failed: it didn’t like that the database 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 and everything succeeded.

```
$ 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

Installing Heroku CLI deployment plugin

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

Creating Heroku deployment files

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

Building application

...

BUILD SUCCESSFUL

Total time: 50.636 secs

Deploying application

Uploading your application code.  
This may take several minutes depending on your connection speed...  
Uploading 21-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.

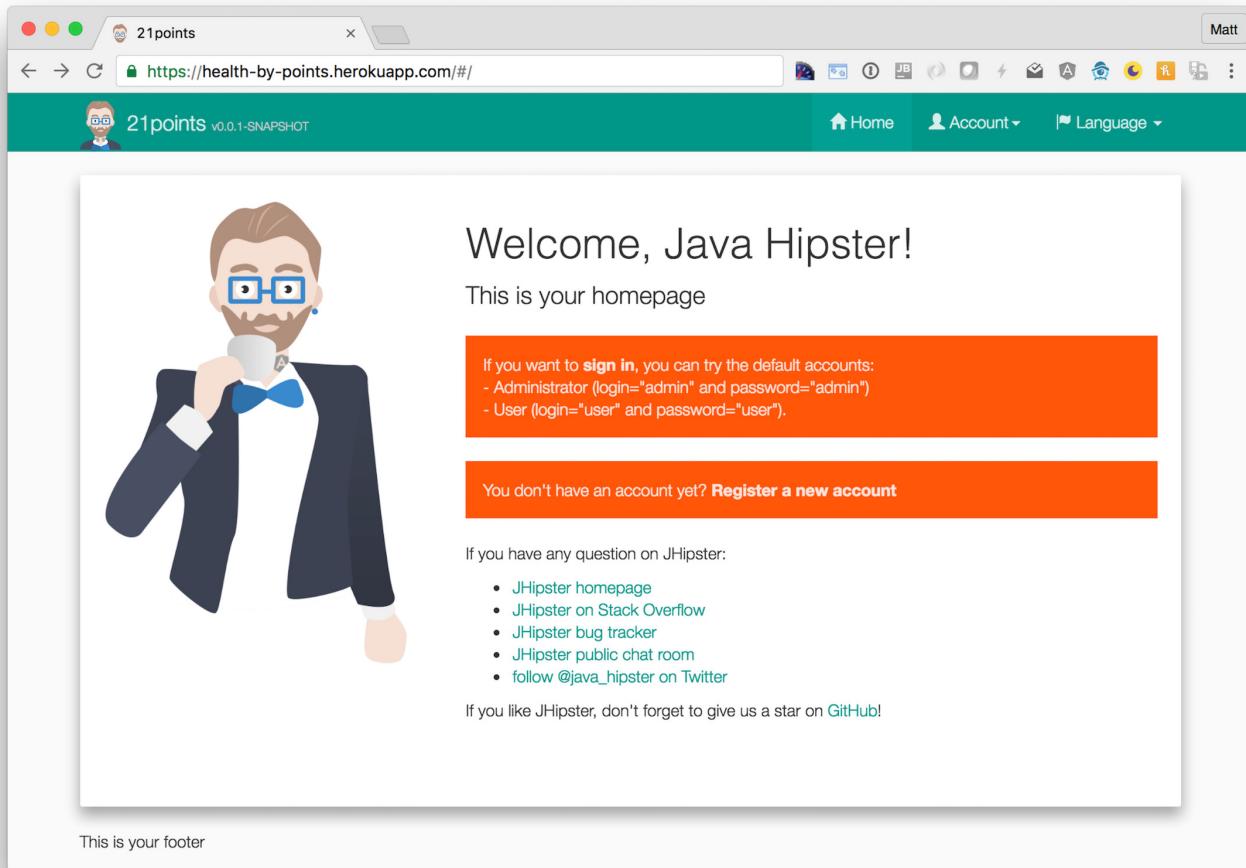


Figure 12. First deployment to Heroku

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-deploy](#). Use the following command to install this plugin.

```
heroku plugins:install https://github.com/heroku/heroku-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 deploy:jar --jar build/libs/*war --app health-by-points
```

With Maven, the commands look slightly different:

```
mvn install -Pprod -DskipTests
heroku deploy:jar --jar target/*.war
```

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

```
$ heroku deploy:jar --jar build/libs/*war --app health-by-points
Uploading build/libs/21points-0.1-SNAPSHOT.war....
----> Packaging application...
  - app: health-by-points
  - including: build/libs/21points-0.1-SNAPSHOT.war
----> Creating build...
  - file: slug.tgz
  - size: 63MB
----> Uploading build...
  - success
----> Deploying...
remote:
remote: ----> Fetching custom tar buildpack... done
remote: ----> JVM Common app detected
remote: ----> Installing OpenJDK 1.8... done
remote: ----> Discovering process types
remote:       Procfile declares types -> web
remote:
remote: ----> Compressing... done, 112.5MB
remote: ----> Launching... done, v14
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. I was soon disappointed when the application didn't start within 60 seconds.

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

This is an expected problem with JHipster and Heroku. I created a support ticket at <https://help.heroku.com/> and asked to increase my application's allowed timeout to 120 seconds. Heroku's support team was quick to respond and boosted my timeout within minutes.



If you need to reset your Postgres database on Heroku, you can do so by logging into <http://api.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.

```
2015-08-20T14:37:54.660329+00:00 app[web.1]: Caused by:  
org.elasticsearch.client.transport.NoNodeAvailableException:  
None of the configured nodes are available: []
```

I searched for an Elasticsearch add-on for Heroku and found [Bonsai Elasticsearch](#). Its cheapest plan cost \$10/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.

```
2015-08-20T15:11:36.809174+00:00 heroku[web.1]: Process running mem=561M(109.6%)  
2015-08-20T15:11:36.809174+00:00 heroku[web.1]: Error R14 (Memory quota exceeded)  
2015-08-20T15:11:41.395945+00:00 heroku[router]: at=info method=POST  
path="/api/register?cacheBuster=1440083497301" host=www.21-points.com ...  
2015-08-20T15:11:43.106106+00:00 app[web.1]: [WARN]  
org.jhipster.health.service.MailService - E-mail could not be sent to  
user 'matt@raibledesigns.com', exception is: Mail server connection failed; nested  
exception is javax.mail.MessagingException:  
Connection error (java.net.ConnectException: Connection refused). Failed messages:  
javax.mail.MessagingException:  
Connection error (java.net.ConnectException: Connection refused)
```



You might notice the “Memory quota exceeded” message in the logs. I receive this often when running JHipster applications under Heroku’s [free and hobby dynos](#). My application stays running, though, so I’ve learned to ignore it.

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
  tls: false
  auth: true
  from: app@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](http://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 deploy:jar --jar 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 -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](#). I followed JHipster's Jenkins 2 documentation and opted for the global Node.js installation.

## Travis CI

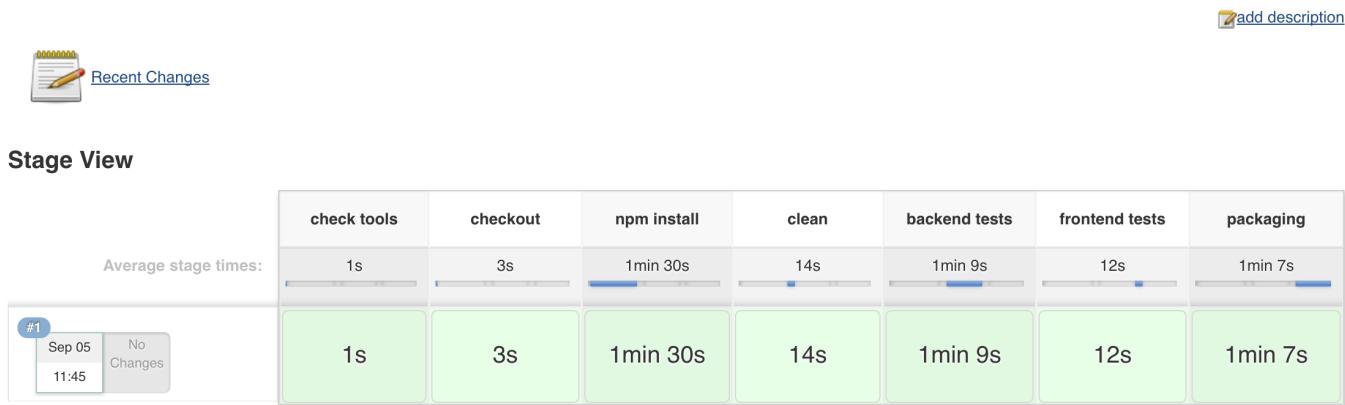
JHipster ships with a `.travis.yml` file that allows you to run your application's tests on [Travis CI](#). You can modify it to run Protractor tests using something like the following:

```
language: java
jdk:
  - oraclejdk8
sudo: false
cache:
  directories:
    - node
    - node_modules
    - $HOME/.gradle
env:
  - NODE_VERSION=4.4.7
before_install:
  - nvm install $NODE_VERSION
  - npm install -g npm
  - npm install -g bower gulp-cli
  - node -v
  - npm -v
  - bower -v
  - gulp -v
  - java -version
install: npm install
script:
  - ./gradlew clean
  - ./gradlew test
  - gulp test
  - sleep 20s
  - export DISPLAY=:99.0
  - sh -e /etc/init.d/xvfb start
  - ./gradlew bootRun &
  - bootPid=$!
  - sleep 120s
  - gulp itest
  - ./gradlew bootRepackage -Pprod -x test
notifications:
  webhooks:
    on_success: change # options: [always|never|change] default: always
    on_failure: always # options: [always|never|change] default: always
    on_start: false    # default: false
```

I created a new job called "21-points" with a Pipeline script from SCM. By default, a JHipster-generated

project contains a [Jenkinsfile](#). I configured a “Poll SCM” build trigger with a schedule of `H/5 * * * *`. After saving the job, I confirmed it ran successfully.

## Pipeline 21-points



### Permalinks

- [Last build \(#1\), 7 min 24 sec ago](#)
- [Last stable build \(#1\), 7 min 24 sec ago](#)
- [Last successful build \(#1\), 7 min 24 sec ago](#)
- [Last completed build \(#1\), 7 min 24 sec ago](#)

Figure 13. Jenkins build #1

I modified this file to add the running of Protractor tests and deploying to Heroku. I checked in my changes to trigger another build.

### Jenkinsfile

```

node {
    // uncomment these 2 lines and edit the name 'node-4.4.7' according to what you
    choose in configuration
    // def nodeHome = tool name: 'node-4.4.7', type:
    'jenkins.plugins.nodejs.tools.NodeJSInstallation'
    // env PATH = "${nodeHome}/bin:${env PATH}"

    stage('check tools') {
        sh "node -v"
        sh "npm -v"
        sh "bower -v"
        sh "gulp -v"
    }

    stage('checkout') {
        checkout scm
    }

    stage('npm install') {
        sh "npm install"
    }
}

```

```

stage('clean') {
    sh "./gradlew clean"
}

stage('backend tests') {
    sh "./gradlew test"
}

stage('frontend tests') {
    sh "gulp test"
}

stage('protractor tests') {
    sh '''./gradlew &
bootPid=$!
sleep 45s
gulp itest
kill $bootPid
'''
}

stage('packaging') {
    sh "./gradlew bootRepackage -Pprod -x test"
}

stage('deploying') {
    sh "heroku deploy:jar --jar build/libs/*.war --app health-by-points"
}
}

```

After making these changes, I found out that my Protractor tests were failing. I tried running them using `gulp itest` and found that Firefox was not loading the application. Since I'd recently experienced Firefox and Protractor issues on another project, I changed to use Chrome. To do this, I modified `protractor.conf.js` and fixed the path to the Chrome driver.

*src/test/javascript/protractor.conf.js*

```

@@ -5,7 +5,7 @@ var prefix = 'src/test/javascript/'.replace(/[^/]+/g,'..');

exports.config = {
  seleniumServerJar: prefix + 'node_modules/protractor/node_modules/webdriver-
manager/selenium/selenium-server-standalone-2.53.1.jar',
-  chromeDriver: prefix + 'node_modules/protractor/selenium/chromedriver',
+  chromeDriver: prefix + 'node_modules/protractor/node_modules/webdriver-
manager/selenium/chromedriver_2.22',
  allScriptsTimeout: 20000,

  suites: {
@@ -15,7 +15,7 @@ exports.config = {

  capabilities: {
-    'browserName': 'firefox',
+    'browserName': 'chrome',
    'phantomjs.binary.path': require('phantomjs-prebuilt').path,
    'phantomjs.ghostdriver.cli.args': ['--loglevel=DEBUG']
  },

```

This got me a bit further, but tests still failed. I banged my head against the wall for a few hours before I figured out the problem: `translate` is a reserved attribute name in HTML5 and Chrome (and Safari) return a boolean for this value if it's present. Not only that, but Protractor's `element.getAttribute()` returns a promise rather than a value.

As part of this debugging exercise, I also discovered that Firefox 48 does not work with Protractor. The most recent version I found that works is Firefox 46. Also, Firefox 46 (and below) don't require you to resolve the promise returned from Protractor's `element.getAttribute()`. However, since both Safari and Chrome require this change (and it works in Firefox), I changed all `translate` attributes in the project's `*.html` files to `data-translate` and I changed all Protractor tests (in `src/test/javascript/e2e`) to process the promise and validate the value afterward.

```

element.all(by.css('h1')).first().getAttribute('data-translate').then(function (value) {
  expect(value).toMatch(/home.title/);
});

```

After making these changes, I [created a pull request](#) to fix this in JHipster's main generator. Then I spent hours trying to get all the Protractor tests passing. I discovered that logout wasn't happening consistently for Chrome and Safari, causing some tests to fail. Finally, I switched back to Firefox and was able to get consistent results.

`src/test/javascript/protractor.conf.js`

```
@@ -15,7 +15,7 @@ exports.config = {
  },
  capabilities: {
+   'browserName': 'firefox',
-   'browserName': 'chrome',
   'phantomjs.binary.path': require('phantomjs-prebuilt').path,
   'phantomjs.ghostdriver.cli.args': ['--loglevel=DEBUG']
 },
```

I checked in my changes, restarted Jenkins, and verified that all stages in my pipeline passed.

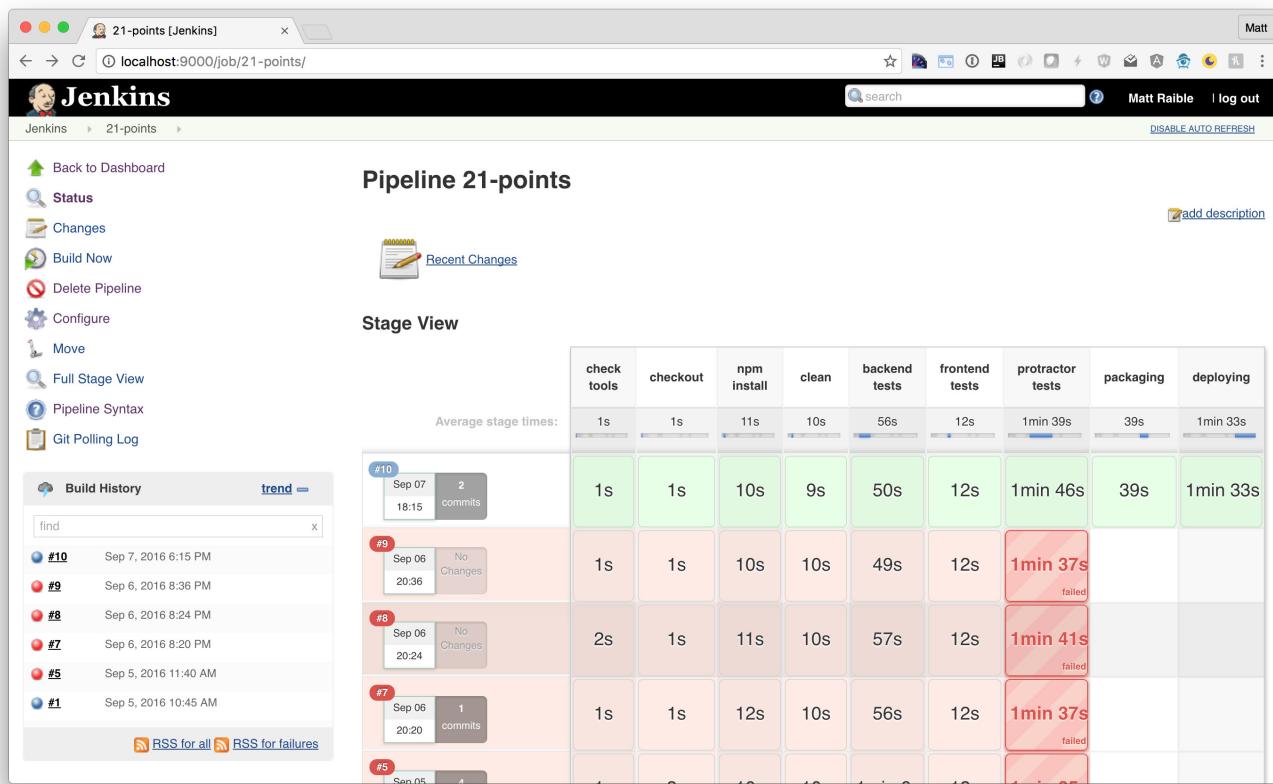


Figure 14. Jenkins success!

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

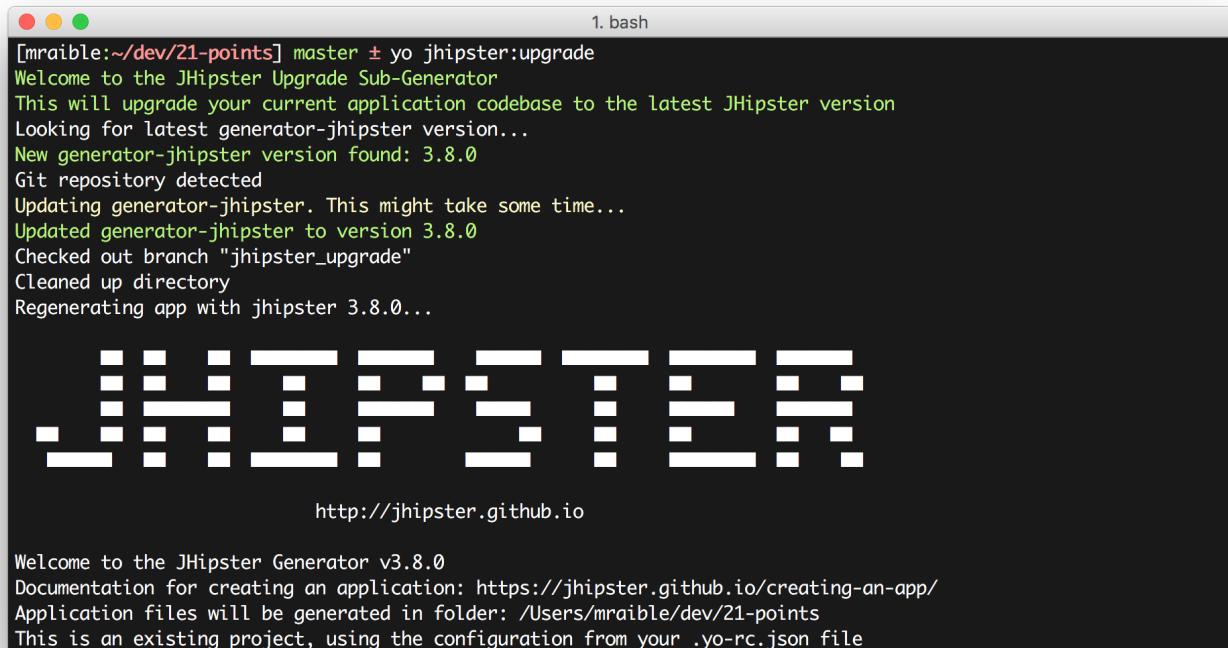
## Source code

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 3.7.0, I wanted to upgrade it to the latest release. Rather than doing it manually like I did previously, I tried the using the [upgrade sub-generator](#). I'd tried it before and it didn't work. Thanks to a [recent bug fix](#), it worked like a charm this time.



A terminal window titled "1. bash" showing the output of a JHipster upgrade command. The output includes the welcome message, generator detection, version update, and a large JHipster logo. It also displays the generator's welcome message for version 3.8.0.

```
[mraible:~/dev/21-points] master ✘ yo jhipster:upgrade
Welcome to the JHipster Upgrade Sub-Generator
This will upgrade your current application codebase to the latest JHipster version
Looking for latest generator-jhipster version...
New generator-jhipster version found: 3.8.0
Git repository detected
Updating generator-jhipster. This might take some time...
Updated generator-jhipster to version 3.8.0
Checked out branch "jhipster_upgrade"
Cleaned up directory
Regenerating app with jhipster 3.8.0...

JHIPSTER

http://jhipster.github.io

Welcome to the JHipster Generator v3.8.0
Documentation for creating an application: https://jhipster.github.io/creating-an-app/
Application files will be generated in folder: /Users/mraible/dev/21-points
This is an existing project, using the configuration from your .yo-rc.json file
```

Figure 15. JHipster Upgrade Start

```
[09:44:20] gulp-inject 24 files into index.html.
[09:44:20] Finished 'inject:test' after 99 ms
[09:44:20] Finished 'inject:vendor' after 67 ms
[09:44:20] Starting 'inject:dep'...
[09:44:20] Finished 'inject:dep' after 6.04 µs
[09:44:20] Starting 'sass'...
[09:44:21] Tested 3 tests, 3 passes, 0 failures: PASS
[09:44:21] Finished 'sass' after 494 ms
[09:44:21] Starting 'copy:languages'...
[09:44:21] Finished 'copy:languages' after 4.68 ms
[09:44:21] Starting 'inject:app'...
[09:44:21] gulp-inject 103 files into index.html.
[09:44:21] Finished 'inject:app' after 121 ms
[09:44:21] Starting 'inject:troubleshoot'...
[09:44:21] gulp-inject 1 files into index.html.
[09:44:21] Finished 'inject:troubleshoot' after 4.36 ms
Successfully regenerated app with jhipster 3.8.0
Committed with message "Generated with JHipster 3.8.0"
Checked out branch "master"
Merging changes back to master...
Merge done !
Please now fix conflicts if any, and commit !
Upgraded successfully.
```

*Figure 16. JHipster Upgrade Result*

After the command finished (in just over two minutes), I fixed the conflicts and checked in my changes. If you’re reading this and notice that 21-Points Health is using a version newer than 3.8.0, it’s likely because I used this handy sub-generator again.

## Summary

This section showed you how I created a health-tracking web application with JHipster. It walked you through upgrading to the latest release of JHipster and how to generate code with `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. AngularJS, Bootstrap, Gulp, 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.

## AngularJS

AngularJS (Angular) is the MVC framework used by JHipster. It's written in JavaScript and just using it makes you a hipster. I've been tracking statistics on jobs and skills for JavaScript MVC frameworks ever since I compared JVM Web frameworks at Devoxx France in 2013. At that time, Backbone was the dominant framework.

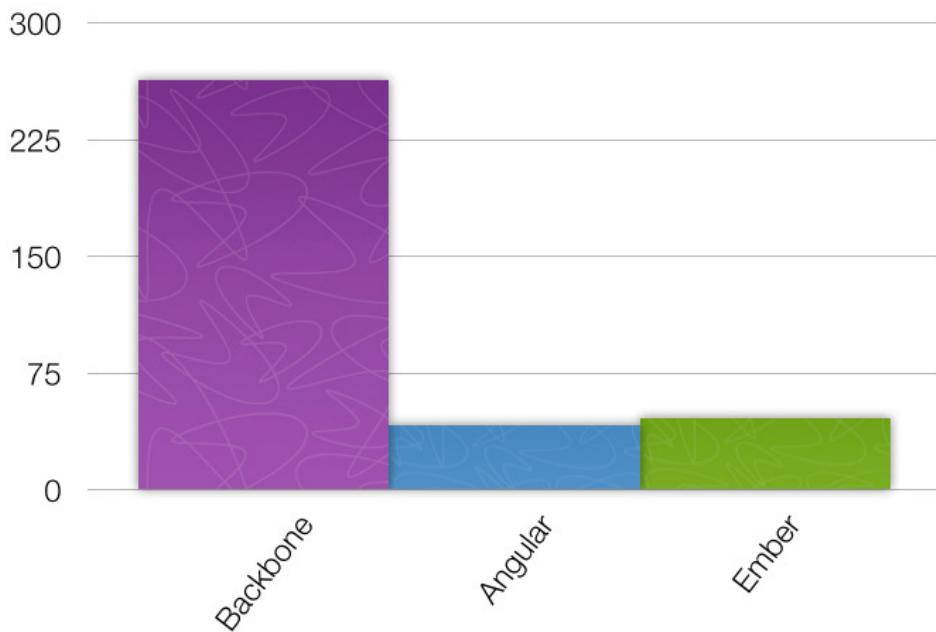


Figure 17. Jobs on Dice, March 2013

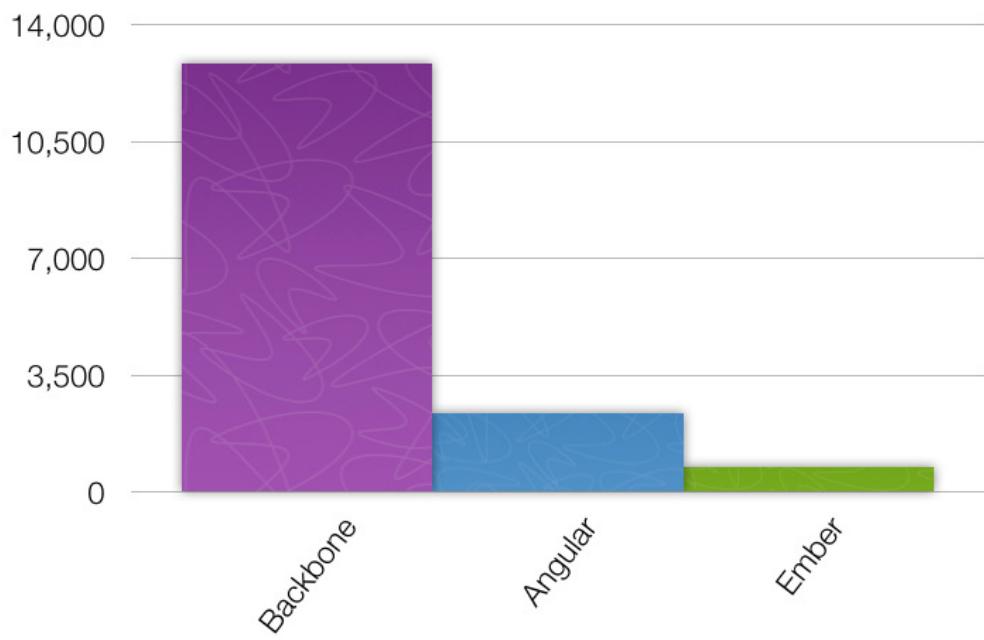


Figure 18. Skills on LinkedIn, March 2013

In February 2014, I updated those statistics for a presentation. The number of Dice jobs that mentioned Angular had grown to roughly equal the number that mentioned Backbone, and a lot of people had added Angular to their LinkedIn profiles. I found that Ember had grown around 300%, Backbone 200%, and Angular 1,000%!

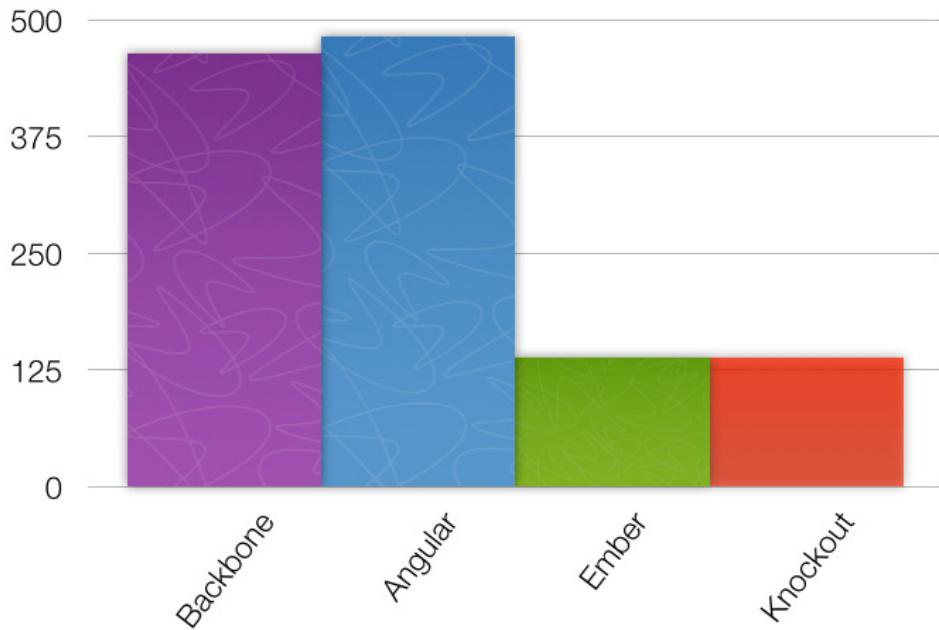
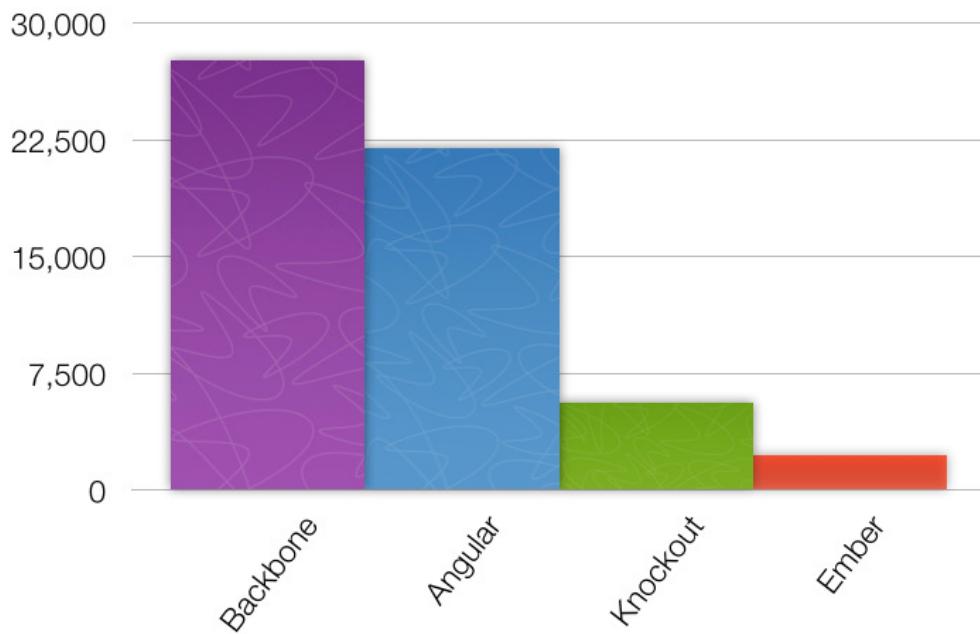


Figure 19. Jobs on Dice, February 2014



*Figure 20. Skills on LinkedIn, February 2014*

In March 2015, I updated these statistics once again. I found the number of jobs for Angular had doubled over the last year, while jobs for Ember and Backbone had fallen slightly. The number of developers who mentioned skills in Ember and Backbone had increased 200%, while the number of skilled Angular folks had risen 400%.

Angular has rapidly become the most-popular MVC web framework. Like Struts in the early 2000s and Rails in the mid-2000s, it's changed the way developers write web applications. Today, data is exposed via REST APIs and UIs are written in JavaScript. 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:

- Controllers: JavaScript functions that retrieve data from services and expose it to templates using `$scope`.
- Services: JavaScript functions 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.
- Filters: 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 Angular in three weeks. At that time, Angular 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 Angular version was only 1,000 lines of code!

## Basics

Creating Hello World with Angular is pretty simple.

```
<!doctype html>
<html ng-app>
<head>
  <title>Hello World</title>
</head>
<body>
<div>
  <label>Name:</label>
  <input type="text" ng-model="name" placeholder="Enter a name here">
  <hr>
  <h1>Hello {{name}}!</h1>
</div>
<script src="http://code.angularjs.org/1.5.8/angular.min.js"></script>
</body>
</html>
```

In this example, `ng-model` maps to the value displayed in `{{name}}`. When using jQuery to do something similar, you have to add an event handler that listens for a change of value. With Angular, the two-way binding is done for you and it saves you from writing a lot of boilerplate code. The `ng-app` directive on line 2 makes this an Angular application.

The MVC pattern is a common one for web frameworks to implement. With Angular, the model is represented by a JavaScript object that you create or retrieve from a service. The view is a HTML template and the controller is a JavaScript function that watches for changes to the model from the user.

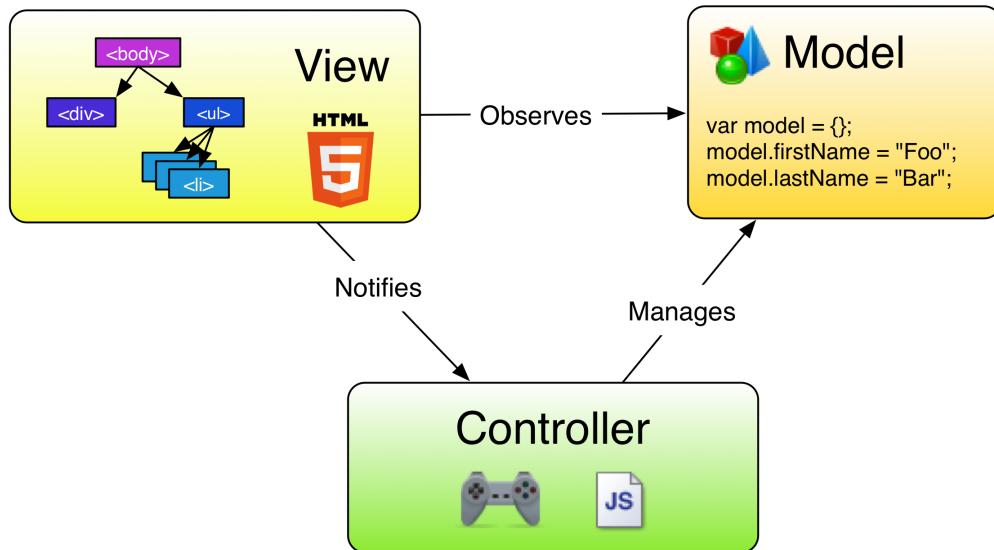


Figure 21. MVC in Angular

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

```

.controller('SearchController', function () {
  console.log("In Search Controller...");
})

.factory('SearchService', function ($http) {
  var service = {
    query: function (term) {
      return $http.get('/api/search/' + term);
    }
  };
  return service;
});
  
```



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 controller available at a URL, you can use Angular's `ngRoute`.

```
angular.module('myApp.search', ['ngRoute'])

.config(['$routeProvider', function ($routeProvider) {
    $routeProvider
        .when('/search', {
            templateUrl: 'search/index.html',
            controller: 'SearchController'
        })
}])
```

The template to call this controller can be as simple as a form with a button.

```
<form ng-submit="search()">
    <input type="search" name="search" ng-model="term">
    <button>Search</button>
</form>
```

To get the results from the `SearchService` in your controller, you can use Angular's dependency injection.

```
.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 `SearchController`, simply add it as a parameter to the controller's argument list.

② `$scope.search()` is a function that's called from the HTML's `<form>`, wired up using the `ng-submit` directive.

③ `$scope.term` is a variable that's wired to `<input>` using the `ng-model` directive.

Another common way to write controllers is to name the arguments passed into the function. This helps if you plan to obfuscate your code.



Angular allows you to bind data to its `$scope` object. It represents your application's model. There's also a `$rootScope` object that can be used to share data between components. Scopes occupy a hierarchical structure that mimics the DOM structure. Scopes can watch expressions and propagate events.



The generated code in JHipster was changed in 2016 to follow [John Papa's Angular Style Guide](#). This means that `vm` (a.k.a the *view model*) is often used instead of `$scope` for exposing data to the view.

```
.controller('SearchController', ['$scope', 'SearchService', function ($scope,
SearchService) {
    ...
}]);
```

To make the aforementioned module work, you need a HTML page that includes Angular and `ngRoute`.

*index.html*

```
<!DOCTYPE html>
<html ng-app="myApp"> ①
<head lang="en">
    <meta charset="UTF-8">
    <title>My AngularJS App</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
<ul class="menu">
    <li><a href="#/search">search</a></li>
</ul>

<div ng-view></div> ②

<script src="http://code.angularjs.org/1.5.8/angular.min.js"></script>
<script src="http://code.angularjs.org/1.5.8/angular-route.min.js"></script>
<script src="app.js"></script> ③
<script src="search/search.js"></script>
</body>
</html>
```

① The `ng-app` directive tells Angular the name of your app.

② `ng-view` is where your rendered template is displayed.

③ `app.js` contains your application's definition.

*app.js*

```
'use strict'; ①

// Declare app level module which depends on views, and components
angular.module('myApp', ['ngRoute', 'myApp.search'])
.config(['$routeProvider', function($routeProvider) {
  // default URL to route to
  $routeProvider.otherwise({redirectTo: '/search'});
}]);
```

- ① ECMAScript 5's strict mode is a way to opt in to a restricted variant of JavaScript. It improves your browser's error reporting and performance.

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 a template with Angular's `ng-repeat` directive.

```
<table ng-show="searchResults.length" class="table">
  <thead>
    <tr>
      <th>Name</th>
      <th>Phone</th>
      <th>Address</th>
    </tr>
  </thead>
  <tbody>
    <tr ng-repeat="person in searchResults">
      <td>{{person.name}}</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) of building a search/edit application with Angular, see my article [Getting Started with AngularJS](#).

## AngularUI Router

[AngularUI Router](#) is a routing component for AngularJS. It's organized around states, to which you can attach routes and other behaviors. Some of its best features include support for nested/named views, tab history, and loading data before rendering a view. You can see an example of its basic features at <http://angular-ui.github.io/ui-router/sample/>.

JHipster includes AngularUI Router (often called ui-router) by default. It's a great framework that I've enjoyed using on several projects. To use it, you'll need to reference its JavaScript file (`angular-ui-router.min.js`) instead of `angular-route.min.js` in `index.html`. You'll also need to change from using `ng-view` to using `ui-view`.

### *index.html*

```
<div ui-view></div>

<script src="http://code.angularjs.org/1.5.8/angular.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-router/0.3.1/angular-ui-router.js"></script>
```

Next, change your `app.js` to use `ui.router` instead of `ngRoute`. Then set up your application states with code like the following listing.

*app.js*

```
myApp.config(function($stateProvider, $urlRouterProvider) {
    // For any unmatched url, redirect to /search
    $urlRouterProvider.otherwise('/search');

    // Set up the states
    $stateProvider
        .state('search', {
            url: '/search',
            templateUrl: 'search/index.html',
            controller: 'SearchController'
        })
        .state('search.edit', {
            url: '/edit/:id',
            templateUrl: 'search/edit.html',
            controller: 'EditController'
        })
    });
});
```

AngularUI Router allows you to look up data before displaying a page. It does this with its `resolve` property. For example, by enhancing the 'search.edit' state, we can look up the person from the `SearchService` before instantiating the controller.

```
.state('search.edit', {
    url: '/edit/:id',
    templateUrl: 'search/edit.html',
    controller: 'EditController',
    resolve: {
        SearchService: 'SearchService', ①
        person: function($stateParams, SearchService) {
            return SearchService.get({id: $stateParams}).$promise; ②
        }
    }
})
```

① This is necessary to indicate the name of the service to the line below this.

② This assumes your service has a `get(id)` method defined.

Once you've done this, you can inject the `person` object into your controller directly.

```
.controller('EditController', function ($scope, person) {
    // make person available to template
    $scope.person = person;
})
```

## Angular's `$resource`

Angular ships with a `$resource factory` that allows you to interact with a [RESTful API](#) with only a few lines of code. If your API is implemented so that you can perform CRUD (create, read, update, delete) at a specific endpoint, you can use `$resource` to wrap that endpoint, and *voilà*, you have a service!

```
.factory('Product', function($resource) {
    return $resource(API_HOSTNAME + '/api/products/:id');
});
```

This gives you a number of actions (or methods) by default. You can easily override these methods or add new ones.

```
{
  'get': {method:'GET'},
  'save': {method:'POST'},
  'query': {method:'GET', isArray:true},
  'remove': {method:'DELETE'},
  'delete': {method:'DELETE'}
};
```

You can then use this service in a controller using `Product.get{id: '123'}`. If you're reading the `id` from the URL, you can use ui-router's `$stateParams` so it becomes `Product.get{id: $stateParams.id}`. To use this service in ui-router's `resolve` property, you'll need to tack on `$promise` to the end.

A `$promise` is a promise of the original server interaction that created this instance or collection. It's a service that helps you run functions asynchronously, and to use their return values (or exceptions) when they are done processing. On success, a promise resolves with the same resource instance or collection object, updated with data from server. On failure, a promise resolves with the HTTP response object, but does not contain a `resource` property. A resource instance also has a `$resolved` property that returns `true` after its first server interaction completes (regardless of success or rejection).

## Angular 2.0

In October 2014, the Angular team announced the [details of 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">
```

Angular 2.x also eliminates the following concepts introduced in 1.0:

- controllers,
- directive definition objects,
- `$scope`,
- `angular.module`, and
- jqLite.

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. The Angular 2.0 project is hosted at <https://angular.io/>. The team plans to support Angular 1.x (at <https://angularjs.org/>) as long as its web traffic exceeds the 2.0 site.

Angular 2.0 was released September 2016. The JHipster team is [currently working on adding support for Angular 2](#). In the meantime, you can learn how to develop with Angular 2 by reading my [Angular 2 Tutorial](#).

Now that you've learned a bit about the hottest web framework 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 its 3.0 version is 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 basic grid system is built with containers, rows, and columns. It's 940 pixels (px) wide, divided into 12 columns. The grid adapts to widths of 724 px and 1170 px, according to your viewport. On viewports narrower than 767 px, the columns become fluid and stack vertically.

A basic example of the grid is shown below.

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

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



*Figure 22. Basic grid on desktop*

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



*Figure 23. Basic grid on a mobile device*

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

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

Bootstrap 3's grid system has four tiers of classes: xs (phones), sm (tablets), md (desktops), and lg (larger desktops). You can use nearly any combination of these classes to create more dynamic and flexible layouts. Below is an example of a grid that's a little more advanced.

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

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

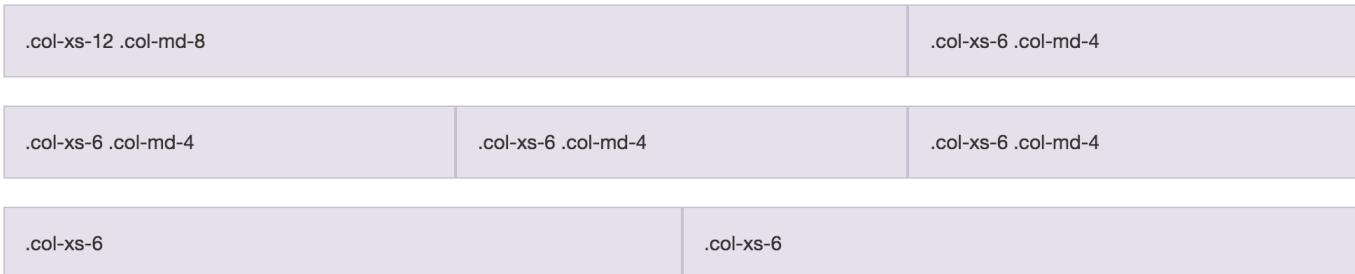


Figure 24. Advanced grid

You can use size indicators to specify breakpoints in the columns. Breakpoints indicate where a column wraps onto the next row. In the HTML above, “md” is the size indicator. Bootstrap supports “xs”, “sm”, “md”, “lg”, and “xl”.

## 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 `.visible-[xs|sm|md|lg]` and `.hidden-[xs|sm|md|lg]`.

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

**Description**
*Figure 25. Basic form element*

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

```
<div class="form-group 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>
```

**Description**

Description is a required field.

*Figure 26. Form element with validation*

## CSS

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

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

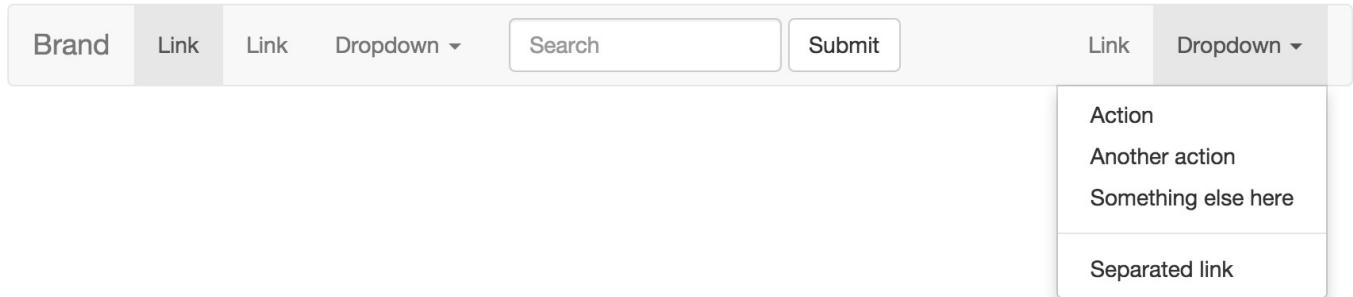
```
<button type="button" class="btn btn-default">Default</button>
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-link">Link</button>
```

*Figure 27. Buttons*

## Components

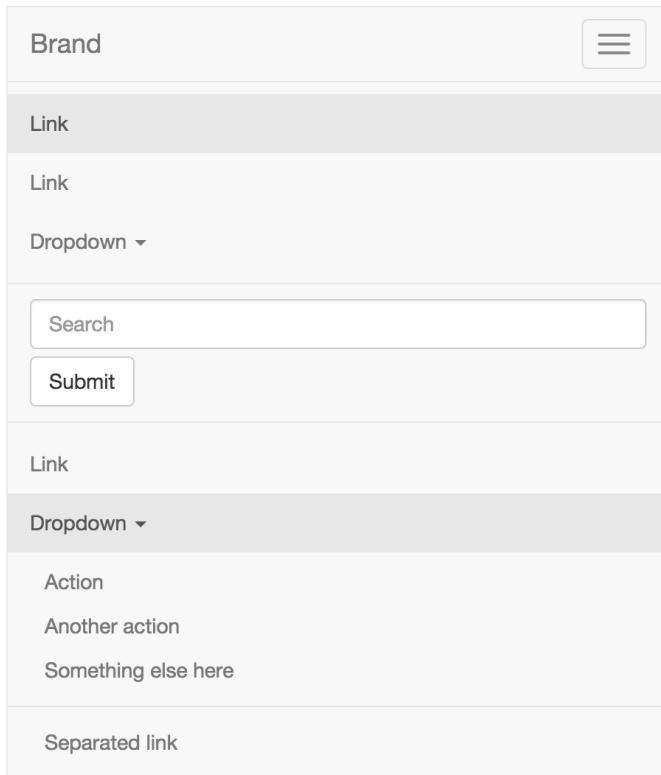
Bootstrap ships with a number of components included. Some require JavaScript; some only require HTML5 markup and CSS classes to work. Its rich set of components have helped make it the most popular project on GitHub. Web developers have always liked components in their frameworks. A framework that offers 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 28. Navbar with dropdown*

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



*Figure 29. Navbar on mobile*

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

```
<nav class="navbar navbar-default" role="navigation">
    <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-target="#navbar-collapse">
            <span class="sr-only">Toggle navigation</span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="#">Brand</a>
    </div>
    <div class="collapse navbar-collapse" id="navbar-collapse">
        <ul class="nav navbar-nav">
            <li class="active"><a href="#">Link</a></li>
            <li><a href="#">Link</a></li>
        </ul>
    </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-success" role="alert">
    <strong>Well done!</strong> You successfully read this important alert message.
</div>
<div class="alert alert-info" role="alert">
    <strong>Heads up!</strong> This alert needs your attention, but it's not super
important.
</div>
<div class="alert alert-warning" role="alert">
    <strong>Warning!</strong> Better check yourself, you're not looking too good.
</div>
<div class="alert alert-danger" role="alert">
    <strong>Oh snap!</strong> Change a few things up and try submitting again.
</div>
```

This renders alerts like the following.

**Well done!** You successfully read this important alert message.

**Heads up!** This alert needs your attention, but it's not super important.

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

**Oh snap!** Change a few things up and try submitting again.

Figure 30. Alerts

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

```
<div class="alert alert-warning alert-dismissible" 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>
```

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



Figure 31. Closeable alert



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

## Icons

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

The [Glyphicons Halflings](#) set of font icons is included with Bootstrap. They're not normally free, but their creator has made them free for Bootstrap users. There are more than 250 glyphs in this set. They're often used to display eye candy on buttons.

```
<button class="btn btn-info"><i class="glyphicon glyphicon-plus"></i> Add</button>
<button class="btn btn-danger"><i class="glyphicon glyphicon-minus"></i> Delete</button>
<button class="btn btn-default"><i class="glyphicon glyphicon-pencil"></i> Edit</button>
```

You can see how the icons change color based on the font color defined for the element they're inside.



Figure 32. Buttons with icons

You can use other icon fonts with Bootstrap and JHipster, [Font Awesome](#) being one of the most popular. It's completely free for commercial use and contains over 500 icons. JHipster uses Font Awesome for its website, but not for generated projects. The [reason](#) for not including Font Awesome is that Glyphicons is the default and it's easy to change the font in your project. Simply add it as a dependency in your `bower.json` file.

## 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/css/main.css` directly. If you're using Sass to compile `*.scss` files, you can modify `src/main/webapp/scss/main.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 `$icon-font-path` variable and imports Bootstrap. Default rules are overridden in `src/main/webapp/scss/main.scss`.

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

```
$icon-font-path: "../fonts/";

/* Sass bower components will be injected here */
/* bower:scss */
@import "../bower_components/bootstrap-sass/assets/stylesheets/_bootstrap.scss";
/* endinject */
/* bower:sass */
/* endinject */
```

## Angular and Bootstrap

There's a number of other open-source components that can help to further enhance an application built with Angular and Bootstrap that are not included in JHipster at the time of this writing:

- [UI-Bootstrap](#): Bootstrap components written in pure AngularJS by the AngularUI Team. No jQuery required.
- [Ionic](#): Offers a library of mobile-optimized HTML, CSS, and JavaScript components, gestures, and tools for building highly interactive apps. Built with Sass and optimized for AngularJS. Provides a native look and feel.
- [AngularJS-Toaster](#): An AngularJS port of the [Toastr](#) non-blocking notification jQuery library.

# 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. [angular-translate](#) provides directives that make it easy to translate your application into multiple languages.

To use angular-translate, you simply add a “data-translate” attribute with a key.

```
<label data-translate="user.firstname">First Name</label>
```

The key references a JSON document, which will return the translated string. AngularJS 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 26 languages. You can also add a new language. To set the default language, modify [src/main/webapp/app/blocks/config/translation.config.js](#) and its preferred-language line.

[src/main/webapp/app/blocks/config/translation.config.js](#)

```
$translateProvider.preferredLanguage('de');
```

## 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 over-qualified 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 is written with [Less](#), a CSS pre-processor with similar features to Sass. However, its founders have [announced](#) that they'll be using SCSS for the 4.0 version.

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

## Gulp

JHipster uses [Gulp](#) for building the client. It used to support [Grunt](#), but it was removed when Gulp's popularity skyrocketed. Gulp allows you to perform tasks like minification, concatenation, compilation (e.g. from TypeScript/CoffeeScript to JavaScript), unit testing, and more.

Gulp is the de facto standard of JavaScript build tools and has over 2,500 [plugins](#). You can install and manage Gulp plugins via [npm](#), a package manager for [Node.js](#). You can install anything specified as a dependency in your project's `package.json` using `npm install`. To install Gulp globally (so you can use it anywhere), run `npm install -g gulp-cli`.

Gulp calls itself “the streaming build system”. Gulp is all about streams and building complex pipelines with ease. It uses Node.js’s streams, and executes faster than Grunt, since it does not open/close files or create intermediary copies. Gulp was released a couple of years after Grunt, and has received a lot of praise from the JavaScript community. In accordance with the UNIX philosophy, Gulp plugins each try to do one thing well.

To show a simple example of how to use Gulp, I integrated it into the angular-seed project. Below are the `package.json` and `gulpfile.js` that optimize everything for production.

The following steps show how to integrate Gulp into [angular-seed](#). This particular build file optimizes your JavaScript, CSS, and HTML for production. It concatenates your JavaScript and CSS files, minifies them, then replaces the references in the HTML to point to the optimized files. First, you have to configure `package.json` to include Gulp and its plugins for web optimization.

#### `package.json`

```
{  
  "name": "gulp-example",  
  "description": "A Gulp example",  
  "version": "1.0.0",  
  "devDependencies": {  
    "gulp": "~3.9.1",  
    "gulp-clean": "~0.3.2",  
    "gulp-concat": "~2.6.0",  
    "gulp-load-plugins": "~1.3.0",  
    "gulp-minify-css": "~1.2.4",  
    "gulp-rev": "~7.1.2",  
    "gulp-uglify": "~2.0.0",  
    "gulp-usemin": "~0.3.24"  
  }  
}
```

*gulpfile.js*

```

var gulp = require('gulp');
$ = require('gulp-load-plugins')();

gulp.task('clean', function () {
    return gulp.src('dist/')
        .pipe($.clean());
});

gulp.task('copy', ['clean'], function () {
    gulp.src(['**', '!index*.html*', '!js/**', '!lib/**', '!**/*.css'], { cwd: 'app/' })
        .pipe(gulp.dest('dist/.'));
});

gulp.task('usemin', ['copy'], function () {
    return gulp.src('app/index.html')
        .pipe($.usemin({
            css: [$.cleanCSS(), $.rev()],
            js: [$.uglify(), $.rev()]
        }))
        .pipe(gulp.dest('dist/'));
});

// Tell Gulp what to do when we type "gulp" into the terminal
gulp.task('default', ['usemin']);

```

Before the tasks run, the `index.html` file points to individual CSS and JavaScript files. The `<!-- build:*`  
`-->` and `<!-- endbuild -->` are used by the `useminPrepare` task to build the list of files for optimization.

*index.html*

```
<!doctype html>
<html lang="en" ng-app="myApp">
<head>
  <meta charset="utf-8">
  <title>My AngularJS App</title>
  <!-- build:css css/seed.min.css -->
  <link rel="stylesheet" href="css/app.css"/>
  <link rel="stylesheet" href="css/app2.css"/>
  <!-- endbuild -->
</head>
<body>
<ul class="menu">
  <li><a href="#/view1">view1</a></li>
  <li><a href="#/view2">view2</a></li>
</ul>

<div ng-view></div>

<div>Angular seed app: v<span app-version></span></div>

<!-- build:js js/seed.min.js -->
<script src="lib/angular/angular.js"></script>
<script src="lib/angular/angular-route.js"></script>
<script src="js/app.js"></script>
<script src="js/services.js"></script>
<script src="js/controllers.js"></script>
<script src="js/filters.js"></script>
<script src="js/directives.js"></script>
<!-- endbuild -->
</body>
</html>
```

After the build runs, the `dist/index.html` file points to the revved, combined, and minified CSS and JavaScript files.

*dist/index.html*

```
<!doctype html>
<html lang="en" ng-app="myApp">
<head>
  <meta charset="utf-8">
  <title>My AngularJS App</title>
  <link rel="stylesheet" href="css/seed-f050d0dcb3.min.css">
</head>
<body>
<ul class="menu">
  <li><a href="#/view1">view1</a></li>
  <li><a href="#/view2">view2</a></li>
</ul>

<div ng-view></div>

<div>Angular seed app: v<span app-version></span></div>

<script src="js/seed-dd96642634.min.js"></script>
</body>
</html>
```

To learn more about JavaScript build tools, see Mark Dagleish's [Build Wars](#) presentation, delivered at MelbJS in Melbourne, Australia, February 12, 2014. I asked Mark what he thought of these two tools today. He recommended [webpack](#) and reading his more recent "[Block, Element, Modifying Your JavaScript Components](#)" article. Since webpack is not an option for JHipster, I will not cover it in this book.

## 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 to use WebSockets in 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 [TrackerController](#) 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.controller.js*

```
(function () {
    'use strict';

    angular
        .module('blogApp')
        .controller('JhiTrackerController', JhiTrackerController);

    JhiTrackerController.$inject = ['$cookies', '$http', 'JhiTrackerService'];

    function JhiTrackerController ($cookies, $http, JhiTrackerService) {
        // This controller uses a Websocket connection to receive user activities in real-time.
        var vm = this;

        vm.activities = [];

        JhiTrackerService.receive().then(null, null, function(activity) {
            showActivity(activity);
        });

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

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

*src/main/webapp/scripts/components/tracker/tracker.service.js*

```
(function() {
    'use strict';
    /* globals SockJS, Stomp */
```

```

angular
  .module('blogApp')
  .factory('JhiTrackerService', JhiTrackerService);

  JhiTrackerService.$inject = ['$rootScope', '$window', '$cookies', '$http', '$q',
  'AuthServerProvider'];

  function JhiTrackerService ($rootScope, $window, $cookies, $http, $q,
  AuthServiceProvider) {
    var stompClient = null;
    var subscriber = null;
    var listener = $q.defer();
    var connected = $q.defer();
    var alreadyConnectedOnce = false;

    var service = {
      connect: connect,
      disconnect: disconnect,
      receive: receive,
      sendActivity: sendActivity,
      subscribe: subscribe,
      unsubscribe: unsubscribe
    };

    return service;
  }

  function connect () {
    //building absolute path so that websocket doesnt fail when deploying with a
    context path
    var loc = $window.location;
    var url = '//' + loc.host + loc.pathname + 'websocket/tracker';
    var authToken = AuthServiceProvider.getToken();
    if(authToken){
      url += '?access_token=' + authToken;
    }
    var socket = new SockJS(url);
    stompClient = Stomp.over(socket);
    var stateChangeStart;
    var headers = {};
    stompClient.connect(headers, function() {
      connected.resolve('success');
      sendActivity();
      if (!alreadyConnectedOnce) {
        stateChangeStart = $rootScope.$on('$stateChangeStart', function () {
          sendActivity();
        });
        alreadyConnectedOnce = true;
      }
    });
  }

  function disconnect () {
    if (stompClient) {
      stompClient.disconnect();
    }
    var promises = [];
    promises.push(connected.promise);
    promises.push(listener.promise);
    promises.push(stateChangeStart);
    $q.all(promises).catch(function (err) {
      console.error(err);
    });
  }

  function receive (message) {
    var data = JSON.parse(message.body);
    if (data.type === 'activity') {
      sendActivity(data);
    }
  }

  function sendActivity (activity) {
    var message = {
      type: 'activity',
      activity: activity
    };
    stompClient.send('/topic/activity', {}, JSON.stringify(message));
  }

  function subscribe (topic) {
    stompClient.subscribe(topic);
  }

  function unsubscribe (topic) {
    stompClient.unsubscribe(topic);
  }
}

```

```
        }
    });
    $rootScope.$on('$destroy', function () {
        if(angular.isDefined(stateChangeStart) && stateChangeStart !== null){
            stateChangeStart();
        }
    });
}

function disconnect () {
    if (stompClient !== null) {
        stompClient.disconnect();
        stompClient = null;
    }
}

function receive () {
    return listener.promise;
}

function sendActivity() {
    if (stompClient !== null && stompClient.connected) {
        stompClient
            .send('/topic/activity',
                {},
                angular.toJson({'page': $rootScope.toState.name}));
    }
}

function subscribe () {
    connected.promise.then(function() {
        subscriber = stompClient.subscribe('/topic/tracker', function(data) {
            listener.notify(angular.fromJson(data.body));
        });
    }, null, null);
}

function unsubscribe () {
    if (subscriber !== null) {
        subscriber.unsubscribe();
    }
    listener = $q.defer();
}
}
})();
```

WebSockets on the server side of a JHipster project are implemented with Spring's [WebSocket support](#).

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: It 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: Records your test URLs so you can push them back out to all devices with a single click.
- Remote inspector: Remotely tweak and debug webpages 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": "2.0.0",
  "description": "The JHipster Mini-Book",
  "repository": {
    "type": "git",
    "url": "git@bitbucket.org:mraible/jhipster-book.git"
  },
  "devDependencies": {
    "gulp": "3.9.1",
    "browser-sync": "2.15.0"
  }
}
```

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 JavaScript that looks very similar to what you see above. 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 JavaScript MVC framework called AngularJS. 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 Gulp. 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 MVC 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 webpages 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, the 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.

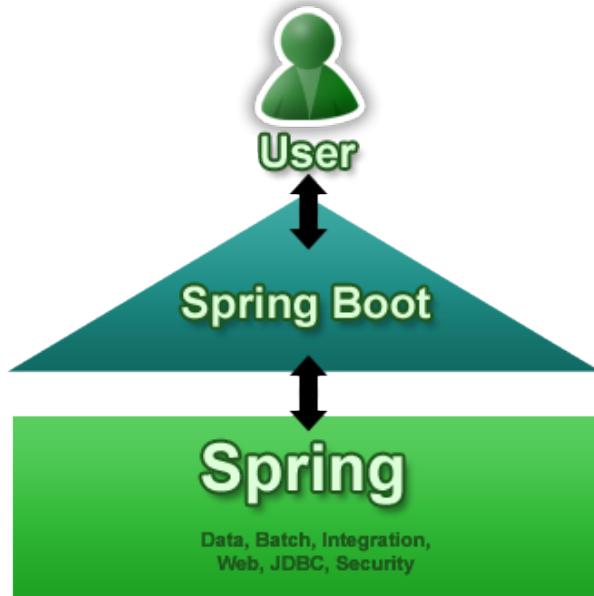


Figure 33. Spring Boot

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`.

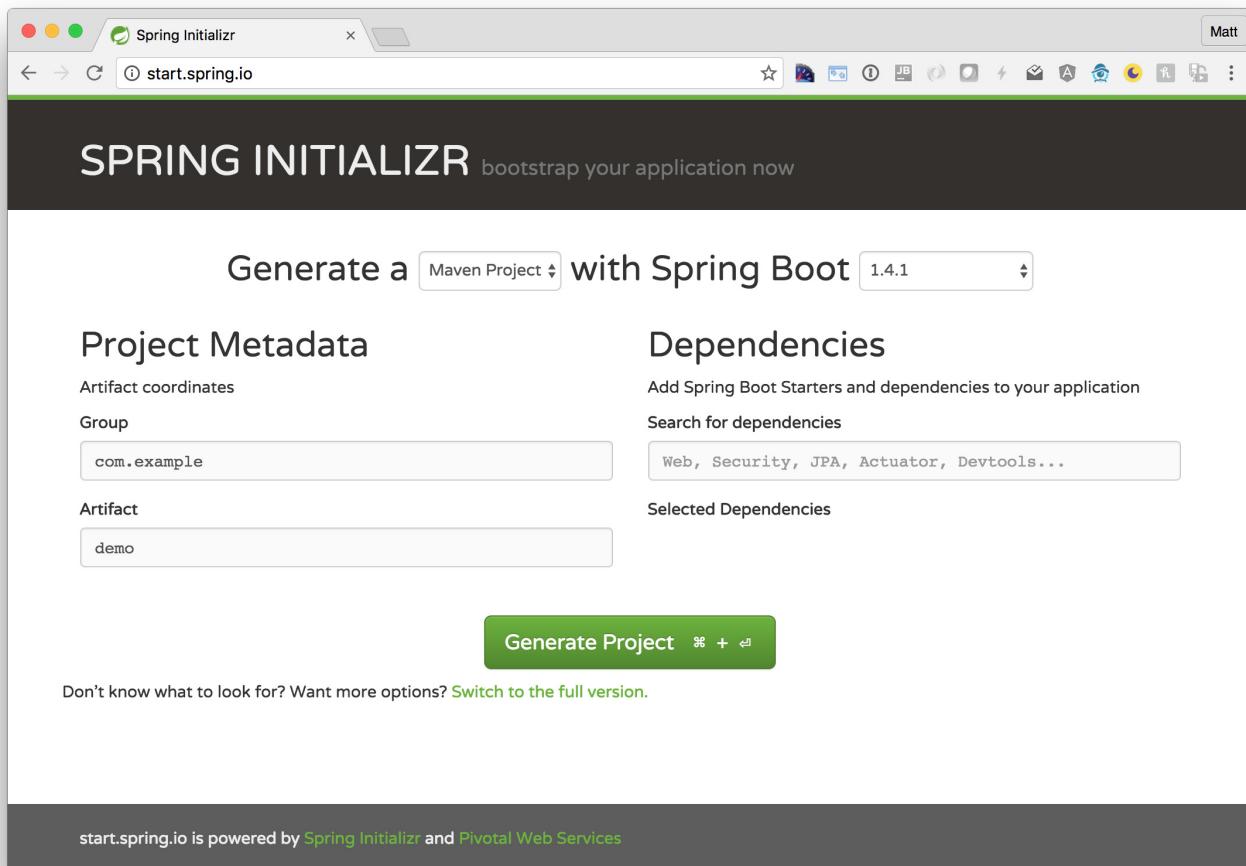
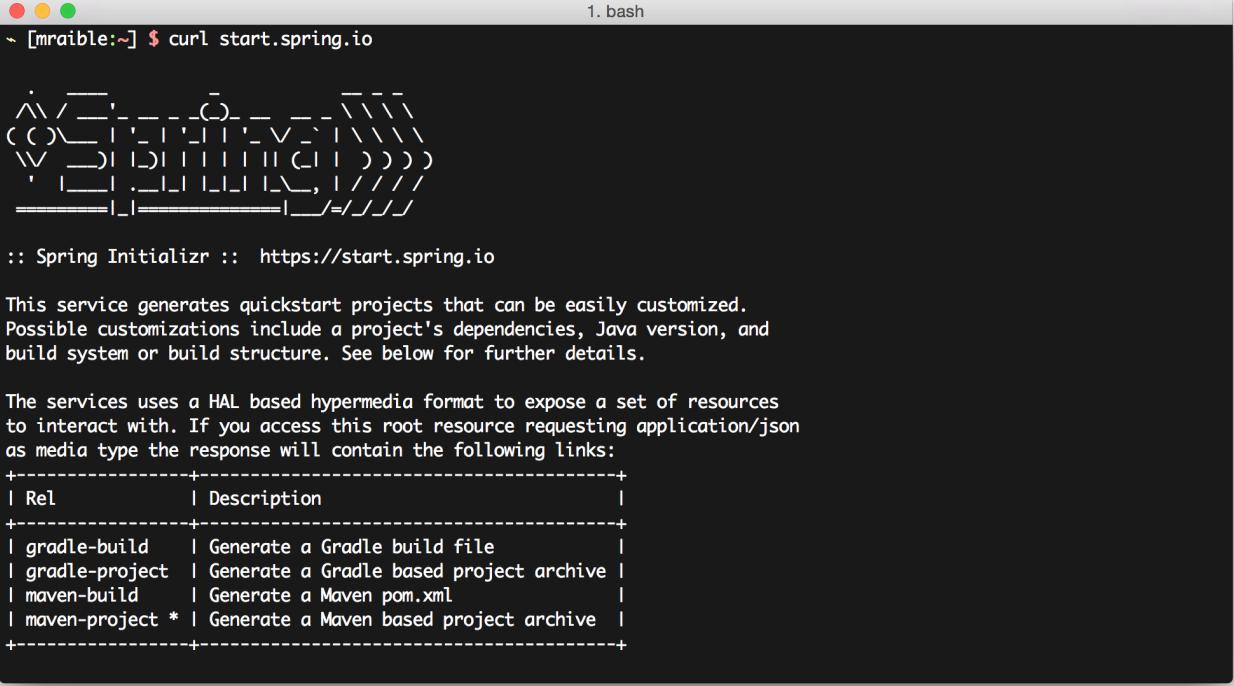


Figure 34. Spring Initializr in a browser



```

1. bash
~ [mraible:~] $ curl start.spring.io

      .--' _ _ _ _ \_ \_ \
     (( \_) | | | | | | \_ \_ \
      \| | | | | | | | | | | |
     _ _ _ _ | | | | | | | |
      _ _ _ _ | | | | | | | |

:: Spring Initializr ::  https://start.spring.io

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

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

```

*Figure 35. Spring Initializr via curl*

Spring Initializr is an Apache 2.0-licensed open-source project that you install and customize to generate Spring projects for your company or team. You can find it on GitHub at <https://github.com/spring-io/initializr>.

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

## Spring CLI

At the bottom of the [start.spring.io](https://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
    │   │           └── DemoApplication.java
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        └── java
            └── com
                └── example
                    └── DemoApplicationTests.java

```

12 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/demo/DemoApplication.java`

```

package 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/DemoApplication.java*

```
@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/demo/DemoApplication.java*

```
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/demo/DemoApplication.java*

```
@Component
class BlogCommandLineRunner implements CommandLineRunner {

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

    @Autowired
    private BlogRepository repository;
}
```

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.

```
2016-10-05 12:02:51.280 INFO 66181 --- [           main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2016-10-05 12:02:51.325 INFO 66181 --- [           main]
o.h.h.i.QueryTranslatorFactoryInitiator : HHH000397: Using ASTQueryTranslatorFactory
[Blog{id=1, name='First'}, Blog{id=2, name='Second'}]
2016-10-05 12:02:51.468 INFO 66181 --- [           main] com.example.DemoApplication
: Started DemoApplication in 9.452 seconds (JVM running for 9.884)
```

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/DemoApplication.java`

```
@RestController
class BlogController {

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

    @Autowired
    BlogRepository repository;
}
```

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. command-line arguments,
2. JNDI attributes from `java:comp/env`,
3. Java system properties (`System.getProperties()`),
4. OS-environment variables,
5. a `RandomValuePropertySource` that only has properties in `random.*`,
6. profile-specific application properties outside of your packaged JAR (`application-{profile}.properties` and YAML variants),

7. profile-specific application properties packaged inside your JAR (`application-{profile}.properties` and YAML variants),
8. application properties outside of your packaged JAR (`application.properties` and YAML variants),
9. application properties packaged inside your JAR (`application.properties` and YAML variants),
10. `@PropertySource` annotations on your `@Configuration` classes, and
11. 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` subdir 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.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.
- `/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.
- `/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-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-websocket
spring-boot-starter-thymeleaf
spring-boot-starter-test
```

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

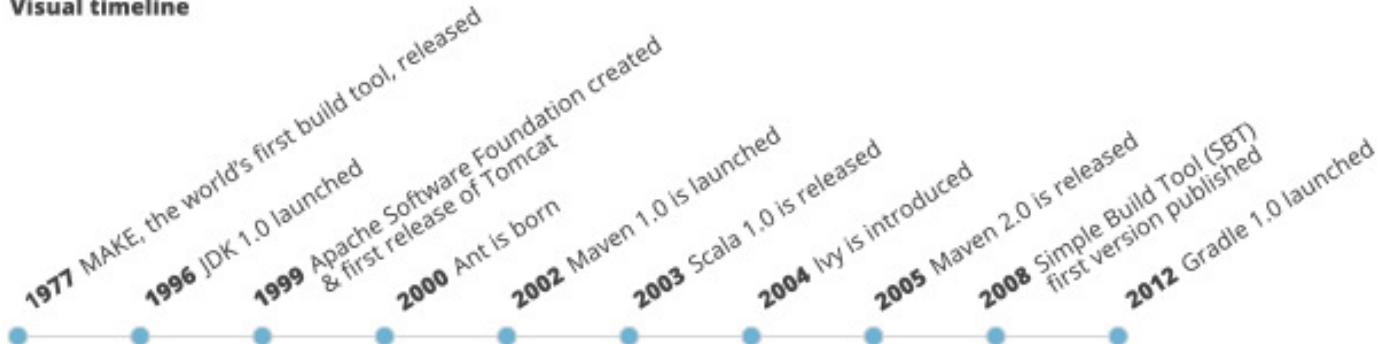
## Maven versus Gradle

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 885 lines of XML. With Gradle, you end up with several `*.gradle` files. In the 21-Points project, their Groovy code adds up to only 441 lines.

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

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

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

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

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

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:translate \
-Dinput=pom.xml -Doutput=pom.${format}
```

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

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

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



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

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

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

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

## IDE support: Running, debugging, and profiling

IDE stands for "integrated development environment". It is the lifeblood of a programmer that 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.

The [Eclipse IDE for Java Developers](#) 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()`, `getUserPrincipal()`, `isUserInRole(role)`, `login(username, password)`, and `logout()`.

JHipster takes the excellence of Spring Security and uses it to provide the real-world authentication mechanism that applications need. When you create a new JHipster project, it provides you with three authentication options:

- **HTTP Session Authentication** — Uses the HTTP session, so it is a stateful mechanism. Recommended for small applications.
- **OAuth2 Authentication** — A stateless security mechanism. You might prefer it if you want to scale your application across several machines.

- **JWT authentication** — Like OAuth2, 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.

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, but you just can't generate a project without an open-source database driver. If you'd like to use a traditional database, select SQL when creating your JHipster project.



JHipster's [Using Oracle](#) guide shows you how to modify a project to support Oracle.

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.

```
? (5/15) Which *type* of database would you like to use? (Use arrow keys)
SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle)
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 OAuth2 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/0000000000000000_initial_schema.xml`

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

Liquibase supports [most major databases](#). If you use MySQL or PostgreSQL, you can use `mvn liquibase:diff` (or `./gradlew 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
```

You could also use [SearchBox Elasticsearch](#). It's an add-on for Heroku that provides hosted, managed, and scalable search with Elasticsearch. It offers a free plan for development and many others to allow scaling up.

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 pre-configured “prod” Maven profile.

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

With Gradle, it's:

```
gradlew -Pprod bootRun
```

The “prod” profile will trigger a [gulp build](#), 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 `dist` 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 AngularJS, 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 Book on AngularJS\*](#) by Ari Lerner (Fullstack.io, December 2013). Lerner has also written *ng-book 2* for Angular 2.

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, expected early 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.

