THE JHIPSTER MINI-BOOK

Matt Raible

InfoQueue
ENTERPRISE SOFTWARE
DEVELOPMENT SERIES

Table of Contents

The JHipster Mini-Book

Published by C4Media, publisher of InfoQ.com.

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

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

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). You will need Node.js and Java installed. The book code was tested against Node.js v0.12 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 a particular part of the code, it's called out with numbers.

```
.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 some topics may be displayed using a sidebar like this.

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 think about this book — what you liked or disliked. Reader feedback helps us develop titles that you get the most out of.

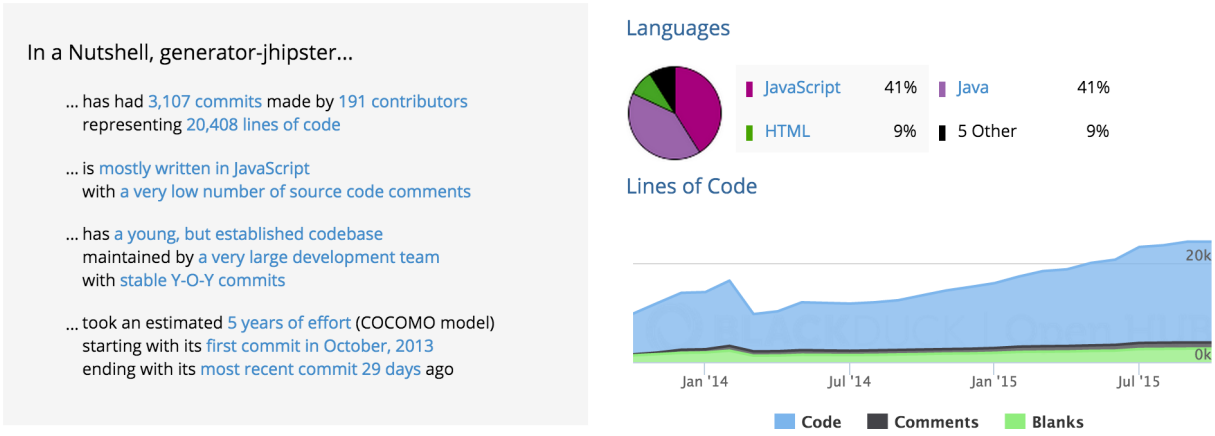
To send us feedback, e-mail us at feedback@infoq.com, send a tweet to @jhipster_book, or post a question on Stack Overflow with the "jhipster-mini-book" tag.

If you have a topic that you have expertise in 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 85 releases! It is an open-source, Apache 2.0-licensed project on GitHub. It has a core team of nine developers and over 150 contributors. You can find its homepage at http://jhipster.github.io. Its Open HUB profile shows it's mostly written in JavaScript and Java.



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 Yeomen 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.)
- 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 is almost 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 Grunt and JHipster.

```
npm install -g grunt-cli
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.

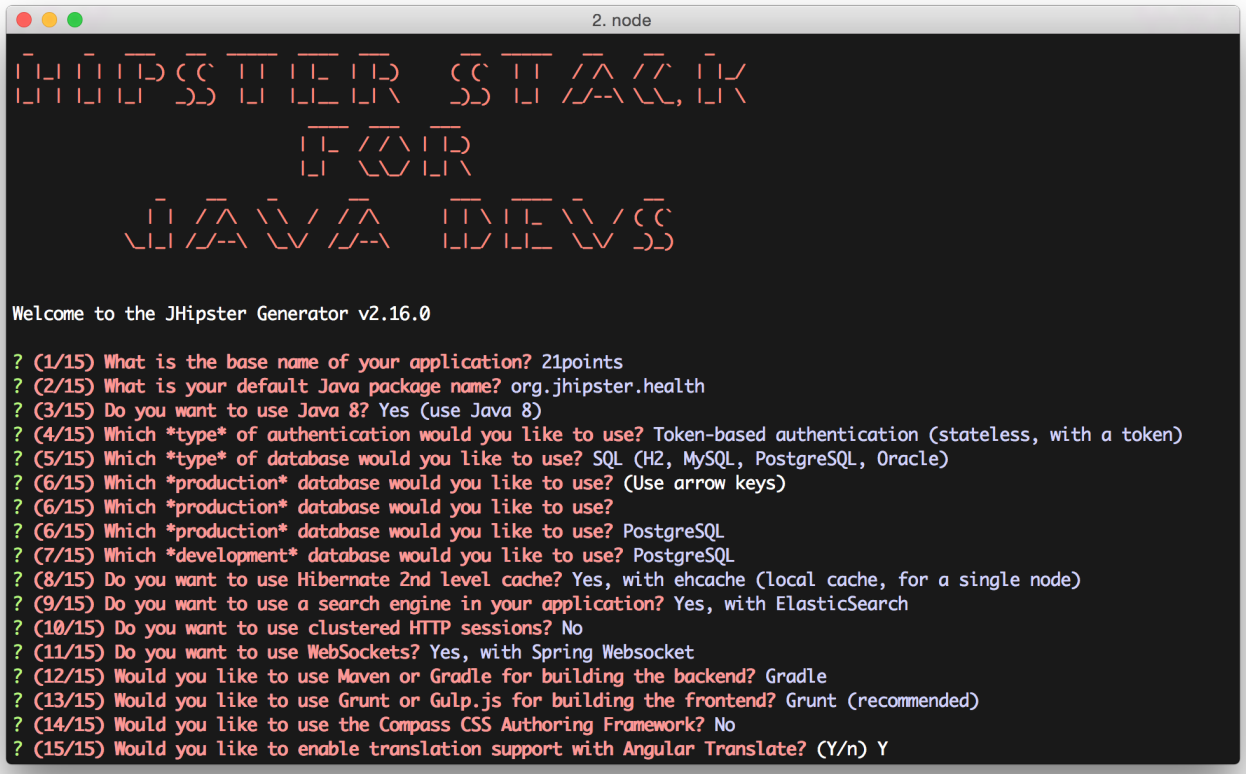


Figure 1. Generating the application

You can see that I chose PostgreSQL as my development and production database. I did this because using a non-embedded database (like H2) 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 (272 in this case), then ran `npm install` followed by `bower install`. To prove everything was good to go, I ran the unit tests using `grunt test`.

Next, I installed Postgres.app and tried creating a local PostgreSQL database. You can see that PostgreSQL didn't like that my database name started with a number.

```
'/Applications/Postgres.app/Contents/Versions/9.4/bin'/psql -p5432
[mraible:~] $ '/Applications/Postgres.app/Contents/Versions/9.4/bin'/psql -p5432psql
(9.4.0)
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 '21points';
                    ^
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-dev.yml` to use this name and the specified credentials.

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

```
     datasource:
         dataSourceClassName: org.postgresql.ds.PGSimpleDataSource
-        url:
-        databaseName: 21points
-        serverName: localhost
-        username: 21points
-        password:
+        url: jdbc:postgresql://localhost/health
+        username: health
+        password: health
```

Adding source control

One of the first things I like to do when creating a new project is to add it to a version-control system (VCS). In this particular case, I chose Git and Bitbucket. The following commands show how I initialized Git, committed the project, added a reference to the remote Bitbucket repository, then pushed everything.

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

$ git add -A

$ git commit -m "Initial checkin of 21-points application"
[master (root-commit) c20f856] Initial checkin of 21-points application
 274 files changed, 13179 insertions(+)
 ...

$ git push origin master
Counting objects: 382, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (353/353), done.
Writing objects: 100% (382/382), 242.01 KiB | 0 bytes/s, done.
Total 382 (delta 55), 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's 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 Building and Deploying a JHipster App with Jenkins.

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 and found a Material Design theme for Bootstrap. To install it, I executed the following command.

```
bower install bootstrap-material-design --save
```

After this completed, I ran `grunt wiredep` to add the new CSS and JavaScript dependencies to `src/main/webapp/index.html`. The wiredep task updates files that refer to Bower dependencies for you. In this case, `src/main/webapp/index.html` and `src/test/javascript/karma.conf.js`.

I followed the theme's Getting Started guide and added the following initialization code to the bottom of the page.

src/main/webapp/index.html

```
<script>
    $.material.init()
</script>
```

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

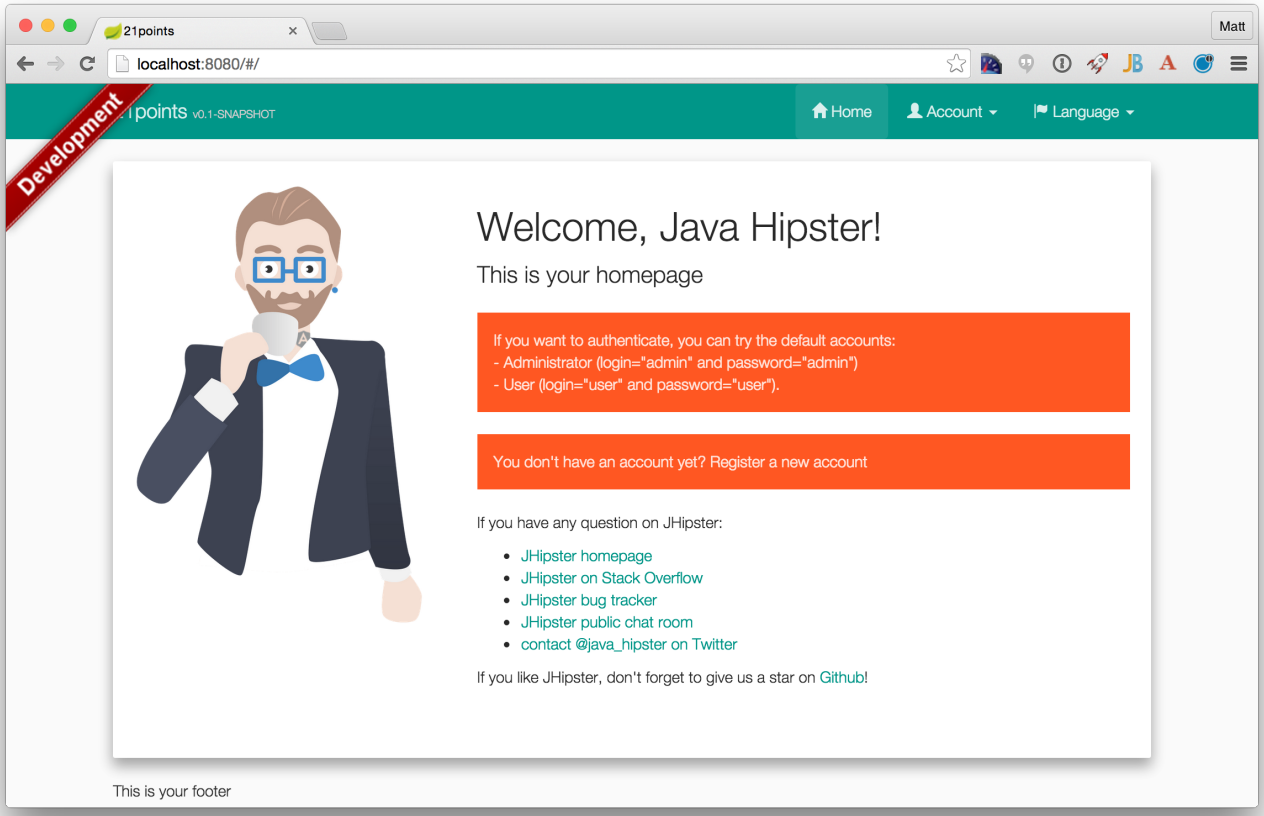


Figure 2. Material Design for Bootstrap theme

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 2.16.0. The latest release as of writing is 2.19.0, so I updated my version with the following command.

```
npm update -g generator-jhipster
```

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 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 -g generator-jhipster`.

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

Remove the file - src/test/javascript/spec/app/account/health/healthControllerSpec.js
Remove the file - src/test/javascript/spec/app/account/login/loginControllerSpec.js
Remove the file - src/test/javascript/spec/app/account/password/passwordControllerSpec.js
Remove the file - src/test/javascript/spec/app/account/password/passwordDirectiveSpec.js
Remove the file - src/test/javascript/spec/app/account/sessions/sessionsControllerSpec.js
Remove the file - src/test/javascript/spec/app/account/settings/settingsControllerSpec.js
Remove the file - src/test/javascript/spec/components/auth/authServicesSpec.js
 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. Most changes were welcome but I wanted to keep my theme changes so I had to add the following back into `bower.json` and run `bower install` again.

bower.json

```
"bootstrap-material-design": "~0.3.0"
```

I still needed to manually restore the call to initialize the Material Design theme at the bottom of `index.html`.

src/main/webapp/index.html

```
<script>
    $.material.init()
</script>
```

I ran `grunt serve` to verify that everything looked good, then committed my updated project to Git.

After integrating the Material Design theme, I deployed to Heroku for the first time. This is covered in the Continuous integration and deployment 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 `JpaRespository` 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. At the time of writing, it does not support generating UI tests. (See issue #897 for why it does not support UI testing.) 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 two 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. JHipster UML is an alternative for those that like visual tools. The supported UML editors include Modelio, UML Designer, GenMyModel and Visual Paradigm. Because I believe the entity sub-generator is simpler to use, I chose that for this project.

The diagram below shows the data model for this project. A user has a goal, which is tied to metrics and a daily log of activities. The activities could be further abstracted so they're not explicitly exercise, meals, and alcohol, but it's important to start, not to get it right the first time.

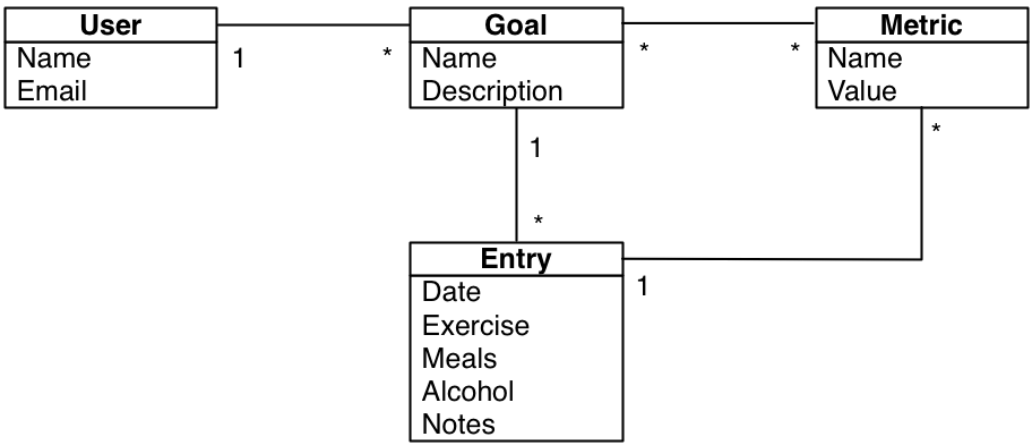


Figure 3. 21-Points Health entity diagram

The most important thing to remember when generating entities with JHipster is that you must generate the entity that owns the relationship first. In this application, the `Metric` entity is owned by `Goal` and `Entry`, so we'll generate that one first. The relationships could be simplified to only track metrics for the entry, but then it'd be difficult to relate that back to the goal and display progress. The following diagram is a simplified version, without a relationship of metrics to goals. For more information, see Managing relationships on the JHipster site.

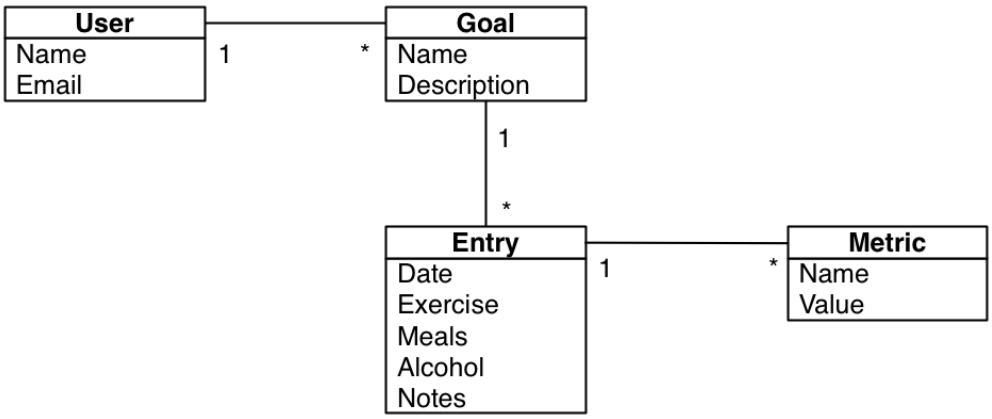


Figure 4. Simple entity diagram

I started by generating a `Goal` entity, with a many-to-one relationship to `User`. Below are the questions and answers I used to generate this entity.

```
$ yo jhipster:entity Goal
The entity Goal is being created.
Generating field #1
? Do you want to add a field to your entity? Yes
? What is the name of your field? name
? What is the type of your field? String
? Do you want to add validation rules to your field? Yes
? Which validation rules do you want to add? Required, Minimum length
? What is the minimum length of your field? 10
================Goal================
name (String) required minlength='10'
Generating field #2
? Do you want to add a field to your entity? Yes
? What is the name of your field? description
? What is the type of your field? String
? Do you want to add validation rules to your field? No
================Goal================
name (String) required minlength='10'
description (String)
Generating field #3
? Do you want to add a field to your entity? No
================Goal================
name (String) required minlength='10'
description (String)
Generating relationships with other entities
? Do you want to add a relationship to another entity? Yes
? What is the name of the other entity? user
? What is the name of the relationship? user
? What is the type of the relationship? many-to-one
? When you display this relationship with AngularJS, which field from 'user' do you want
to use? id
==========Goal=============
name (String)
description (String)
------------------
user - user (many-to-one)
Generating relationships with other entities
? Do you want to add a relationship to another entity? No
==========Goal=============
name (String)
description (String)
------------------
user - user (many-to-one)
? Do you want pagination on your entity? No
```

I didn't add any pagination because I've been tracking my goals quarterly. I may add it after I've been using this app for a while.

After I answered the last question, JHipster generated the files to create/read/update/delete this entity.

```
Everything is configured, generating the entity...
   create .jhipster/Goal.json
   create src/main/java/org/jhipster/health/domain/Goal.java
   create src/main/java/org/jhipster/health/repository/GoalRepository.java
   create src/main/java/org/jhipster/health/repository/search/GoalSearchRepository.java
   create src/main/java/org/jhipster/health/web/rest/GoalResource.java
   create
src/main/resources/config/liquibase/changelog/20150811180009_added_entity_Goal.xml
   create src/main/webapp/scripts/app/entities/goal/goals.html
   create src/main/webapp/scripts/app/entities/goal/goal-detail.html
   create src/main/webapp/scripts/app/entities/goal/goal.js
   create src/main/webapp/scripts/app/entities/goal/goal.controller.js
   create src/main/webapp/scripts/app/entities/goal/goal-detail.controller.js
   create src/main/webapp/scripts/components/entities/goal/goal.service.js
   create src/main/webapp/scripts/components/entities/goal/goal.search.service.js
   create src/test/java/org/jhipster/health/web/rest/GoalResourceTest.java
   create src/test/gatling/simulations/GoalGatlingTest.scala
   create src/main/webapp/i18n/en/goal.json
   create src/main/webapp/i18n/fr/goal.json
```

I proceeded to generate the `Metric` entity, with a many-to-many relationship to `Entry`.

When I tried to use `value`, JHipster warned me this was a reserved word in PostgreSQL, so I used `amount` instead.

```
$ yo jhipster:entity Metric
The entity Metric is being created.
Generating field #1
? Do you want to add a field to your entity? Yes
? What is the name of your field? name
? What is the type of your field? String
? Do you want to add validation rules to your field? Yes
? Which validation rules do you want to add? Required, Minimum length
? What is the minimum length of your field? 2
=================Metric=================
name (String) required minlength='2'
Generating field #2
? Do you want to add a field to your entity? Yes
? What is the name of your field? amount
? What is the type of your field? String
```

```
? Do you want to add validation rules to your field? Yes
? Which validation rules do you want to add? Required
================Metric================
name (String) required minlength='2'
amount (String) required
Generating field #3
? Do you want to add a field to your entity? No
================Metric================
name (String) required minlength='2'
amount (String) required
Generating relationships with other entities
? Do you want to add a relationship to another entity? Yes
? What is the name of the other entity? entry
? What is the name of the relationship? entry
? What is the type of the relationship? many-to-many
? Is this entity the owner of the relationship? No
==========Metric==============
name (String)
amount (String)
-------------------
entry - entry (many-to-many)
Generating relationships with other entities
? Do you want to add a relationship to another entity? Yes
? What is the name of the other entity? goal
? What is the name of the relationship? goal
? What is the type of the relationship? many-to-many
? Is this entity the owner of the relationship? No
==========Metric==============
name (String)
amount (String)
-------------------
entry - entry (many-to-many)
goal - goal (many-to-many)
Generating relationships with other entities
? Do you want to add a relationship to another entity? No
==========Metric==============
name (String)
amount (String)
-------------------
entry - entry (many-to-many)
goal - goal (many-to-many)
? Do you want pagination on your entity? Yes, with pagination links
```

Finally, I created `Entry`, with a many-to-one relationship to `Goal` and `Metric`. Rather than showing you all the questions and answers, I'll explain it in simple terms. I made the `date` a `LocalDate` that's required, the individual set point fields as Integers, and made `notes` a String that's not required. JHipster showed me the following output before generating everything.

```
==========Entry==============
date (LocalDate)
exercise (Integer)
meals (Integer)
alcohol (Integer)
notes (String)
------------------
goal - goal (many-to-one)
metric - metric (many-to-many)
? Do you want pagination on your entity? Yes, with infinite scroll
```

To ensure that everything generated correctly, I ran `./gradlew test`. I received numerous failures, many of them looking similar to the following.

```
org.jhipster.health.web.rest.UserResourceTest > testGetExistingUser FAILED
    java.lang.IllegalStateException
        Caused by: org.springframework.beans.factory.BeanCreationException
            Caused by: javax.persistence.PersistenceException
                Caused by: org.hibernate.AnnotationException
```

I opened `build/reports/tests/index.html` to investigate further and found the following error:

```
Caused by: org.hibernate.AnnotationException: mappedBy reference an unknown target entity
property:
  org.jhipster.health.domain.Goal.metrics in org.jhipster.health.domain.Metric.goals
```

I determined this was caused by generating the `Goal` entity without the relationship to `Metric`, so I added the following Java code to `Goal.java` and ran `./gradlew liquibaseDiffChangelog`.

src/main/java/org/jhipster/health/domain/Goal.java

```
@ManyToMany
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
@JoinTable(name = "GOAL_METRIC",
    joinColumns = @JoinColumn(name="goals_id", referencedColumnName="ID"),
    inverseJoinColumns = @JoinColumn(name="metrics_id", referencedColumnName="ID"))
private Set<Metric> metrics = new HashSet<>();

public Set<Metric> getMetrics() {
    return metrics;
}

public void setMetrics(Set<Metric> metrics) {
    this.metrics = metrics;
}
```

I had to update `liquibase.gradle` to use the same datasource settings I had in `application-dev.yaml` before this command worked. After Liquibase completed successfully, I added the generated file to `src/main/resources/config/liquibase/master.xml`.

src/main/resources/config/liquibase/master.xml

```
<include file="classpath:config/liquibase/changelog/20150811124815_changelog.xml"
relativeToChangelogFile="false"/>
```

I then ran `./gradlew test` again. This time, they failed with the following reason:

```
liquibase.exception.DatabaseException: org.h2.jdbc.JdbcSQLException: Table "ENTRY"
already exists
```

At this moment, I realized that Liquibase was diffing against my "dev" database, while my tests were hitting my "test" (H2) database. When I ran Liquibase's diff command, it was looking at my "dev" database, where no tables had yet been created. To solve this, I removed the changelog reference in `master.xml`, commented out the newly added code in `Goal.java`, and ran `./gradlew bootRun` to generate the initial tables in my "dev" database. Of course, this failed with the same `mappedBy reference` error, but my schema did get created and I ran `./gradlew liquibaseDiffChangelog` again. After adding the generated file to `master.xml`, I was pleased to see my tests passed.

```
BUILD SUCCESSFUL

Total time: 51.422 secs
```

I ran `grunt test` to ensure my UI tests were good to go, then fired up the app and tried everything out. The biggest issue I noticed was that when you created a `Goal`, it showed the `id` of the users instead of their names.

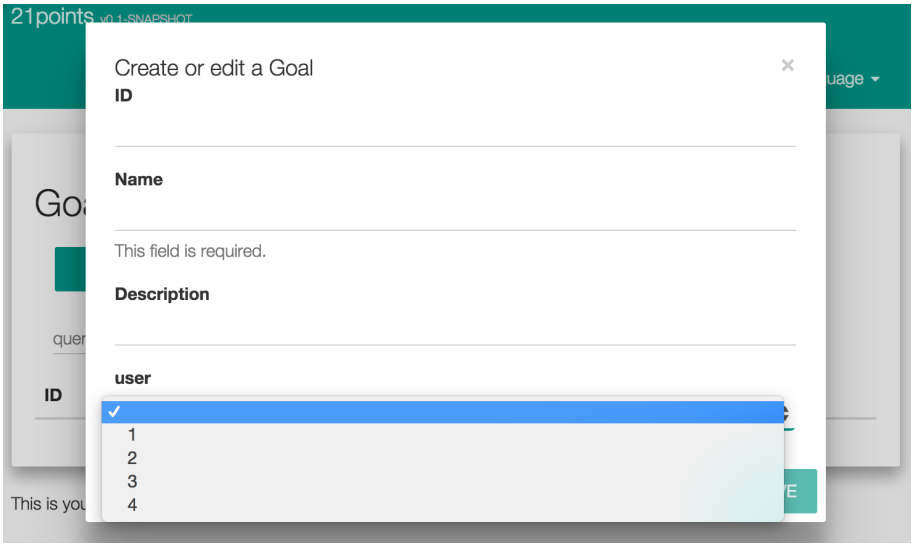


Figure 5. Create a goal with user ID

Since the `id` doesn't provide much information, I changed this to display the user's username instead. In JHipster's `User.java`, this field is called `login`. To make this change, I modified `.jhipster/Goal.json` and changed its `otherEntityField` from having a value of `id` to `login`.

```
"relationships": [
    {
        "relationshipId": 1,
        "relationshipName": "user",
        "relationshipNameCapitalized": "User",
        "relationshipFieldName": "user",
        "otherEntityName": "user",
        "relationshipType": "many-to-one",
        "otherEntityNameCapitalized": "User",
        "otherEntityField": "login"
    }
]
```

After making this change, I ran `yo jhipster:entity goal` to regenerate `Goal.java` and its associated UI. Since I'd modified `Goal.java`, when prompted to overwrite this file, I answered no.

```
 conflict src/main/java/org/jhipster/health/domain/Goal.java
? Overwrite src/main/java/org/jhipster/health/domain/Goal.java? do not overwrite
     skip src/main/java/org/jhipster/health/domain/Goal.java
```

After restarting everything, I was pleased to see the "user" dropdown list contained the `login` field

instead of `id`.

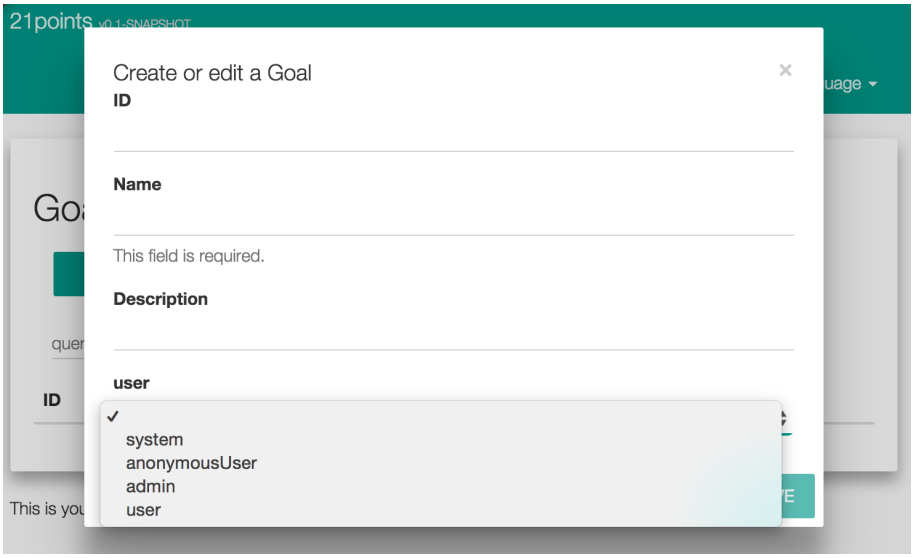


Figure 6. Create a goal with user login

After making this change and regenerating everything, I realized there was an easier way. In `goal-dialog.html`, the following code existed to display the users dropdown list.

src/main/webapp/scripts/app/entities/goal/goal-dialog.html

```
<select class="form-control" id="field_user" name="user" ng-model="goal.user.id" ng-
options="user.id as user.id for user in users">
```

To modify it to display `user.login` instead, I simply needed to change `ng-options` and its *as* expression to the following.

src/main/webapp/scripts/app/entities/goal/goal-dialog.html

```
<select class="form-control" id="field_user" name="user" ng-model="goal.user.id" ng-
options="user.id as user.login for user in users">
```

At this point, I added all the generated files to Git, committed and pushed. I noticed that JHipster had generated 54 files. What a time saver!

I started to play with my newly created app to see if it had the functionality I wanted. I was hoping to easily add daily entries about whether I'd exercised, ate healthy meals, or consumed alcohol. I also wanted to record my weight and blood-pressure metrics when I measured them. When I started using the UI I'd just created, it seemed like it might be able to accomplish these goals, but it also seemed somewhat cumbersome. That's when I decided to create a UI mockup with the main screen and its ancillary screens for data entry. I used OmniGraffle and a Bootstrap stencil to create the following UI mockup.

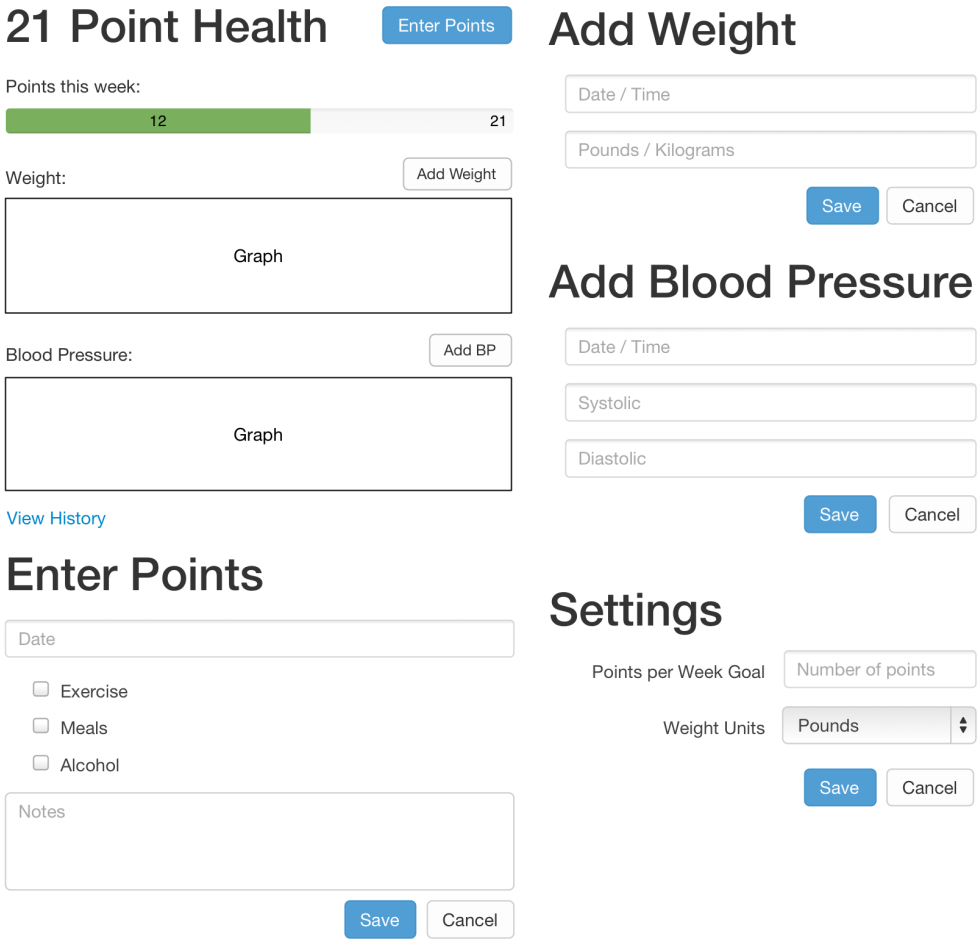


Figure 7. UI mockup

Starting over with a straightforward design

After figuring out how I wanted the UI to look, I realized my data model could be simplified. Before, it was quite generic and could handle a number of metrics. In my new design, I realized I didn't need to track high-level goals (e.g. lose five pounds in Q4 2015). I was more concerned with tracking weekly goals and 21-Points Health is all about how many points you get in a week. I was grateful that JHipster allowed me to quickly see the flaws in my design, then simplify. I created the following diagram as my new data model.

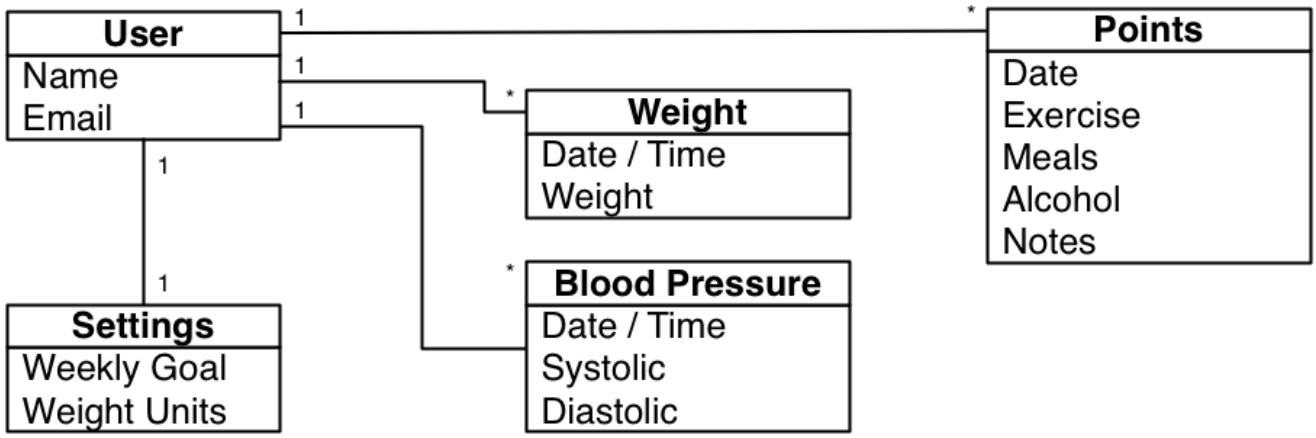


Figure 8. 21-Points Health entity diagram - simplified

JHipster created 54 files when generating the previous data model, REST controllers, and UI. Rather than hunt down all these files and delete them, I reverted to the last commit before them in Git.

If this doesn't work for you, see this Stack Overflow answer for a list of what files to delete.

This is the beauty of a version-control system.

```
$ git reset --hard 8ad48eb
HEAD is now at 8ad48eb Upgraded to JHipster 2.19.0.
```

I also dropped and recreated my local PostgreSQL database.

```
mraible=# drop database health;
DROP DATABASE
mraible=# create database health;
CREATE DATABASE
mraible=# grant all privileges on database health to health;
GRANT
```

Then 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==============
date (LocalDate)
exercise (Integer)
meals (Integer)
alcohol (Integer)
notes (String)
------------------
user - user (many-to-one)
? Do you want to use a Data Transfer Object (DTO)? No, use the entity 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/20150818154309_added_entity_Points.xml
   create src/main/webapp/scripts/app/entities/points/pointss.html
   create src/main/webapp/scripts/app/entities/points/points-detail.html
   create src/main/webapp/scripts/app/entities/points/points-dialog.html
   create src/main/webapp/scripts/app/entities/points/points.js
   create src/main/webapp/scripts/app/entities/points/points.controller.js
   create src/main/webapp/scripts/app/entities/points/points-dialog.controller.js
   create src/main/webapp/scripts/app/entities/points/points-detail.controller.js
   create src/main/webapp/scripts/components/entities/points/points.service.js
   create src/main/webapp/scripts/components/entities/points/points.search.service.js
   create src/test/java/org/jhipster/health/web/rest/PointsResourceTest.java
   create src/test/gatling/simulations/PointsGatlingTest.scala
   create src/main/webapp/i18n/en/points.json
   create src/main/webapp/i18n/fr/points.json
```

I had similar answers for the `Weight` and `BloodPressure` entities. For `Settings`, I created a one-to-one relationship with `User`. I learned that "settings" is a reserved keywork, so used "preferences" instead.

```
$ yo jhipster:entity settings
The entity name cannot contain a JHipster reserved keyword
```

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================
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================
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 received an error about `User` not containing the `preferences` property.

```
Caused by: org.hibernate.AnnotationException: Unknown mappedBy in:
org.jhipster.health.domain.Preferences.user,
referenced property unknown: org.jhipster.health.domain.User.preferences
```

I fixed this by removing the reference to `User` in `Preferences.java`, as well as its `getUser()` and `setUser()` methods.

src/main/java/org/jhipster/health/domain/Preferences.java

```
@OneToOne(mappedBy = "preferences")
@JsonIgnore
private User user;
```

I fixed the relationship by adding a `@OneToOne` mapping in `User.java`:

src/main/java/org/jhipster/health/domain/User.java

```
@OneToOne
@JsonIgnore
private Preferences preferences;

public Preferences getPreferences() {
    return preferences;
}

public void setPreferences(Preferences preferences) {
    this.preferences = preferences;
}
```

I then ran `./gradlew liquibaseDiffChangelog` to generate the changelog and added the XML in the generated file to `*_added_entity_Preferences.xml`.

src/main/resources/config/liquibase/changelog/20150818155956_added_entity_Preferences.xml

```
<!-- Added the preferences field to User -->
<changeSet author="mraible (generated)" id="1439916664921-1">
    <addColumn tableName="JHI_USER">
        <column name="preferences_id" type="int8"/>
    </addColumn>
</changeSet>
<changeSet author="mraible (generated)" id="1439916664921-2">
    <addForeignKeyConstraint baseColumnNames="preferences_id" baseTableName="JHI_USER"
                             constraintName="FK_1r5e40mq4hwtlyd9lemghc8su"
                             deferrable="false" initiallyDeferred="false"
                             referencedColumnNames="id"
                             referencedTableName="PREFERENCES"/>
</changeSet>
```

I had to modify the datasource settings in `liquibase.gradle` again since `git reset` reverted that change.

When I ran `./gradlew test`, I saw some failures, but these were for old tests that I'd already deleted. I opened a ticket with the JHipster project to track this issue.

I checked in six changed files and 78 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 Deploying to Heroku section of this chapter.

Fixed issues with entity and variable names

Shortly after generating all the UI code, I discovered that using plural entity names (e.g. `Points` and `Preferences`) causes you to end up with files, URLs, and variable names that end in two "s" characters. For example, the URL to the points list was `pointss` instead of the more approriate `points`. I fixed this manually in my project and created a bug for JHipster on GitHub.

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, 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:

1. Improved layout of tables and buttons.
2. Improved titles and button labels by editing generated JSON files in `src/main/webapp/i18n/en`.
3. Formatted dates using AngularJS's date filter (for example: `{{bloodPressure.timestamp | date: 'short'}}`).
4. Improved dialogs to hide ID when creating a new entity.
5. Defaulted to current date on new entries.
6. Replaced point metrics with icons on list/detail screens.
7. Replaced point metrics with checkboxes on dialog screen.
8. Added loading indicator for state transitions.

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

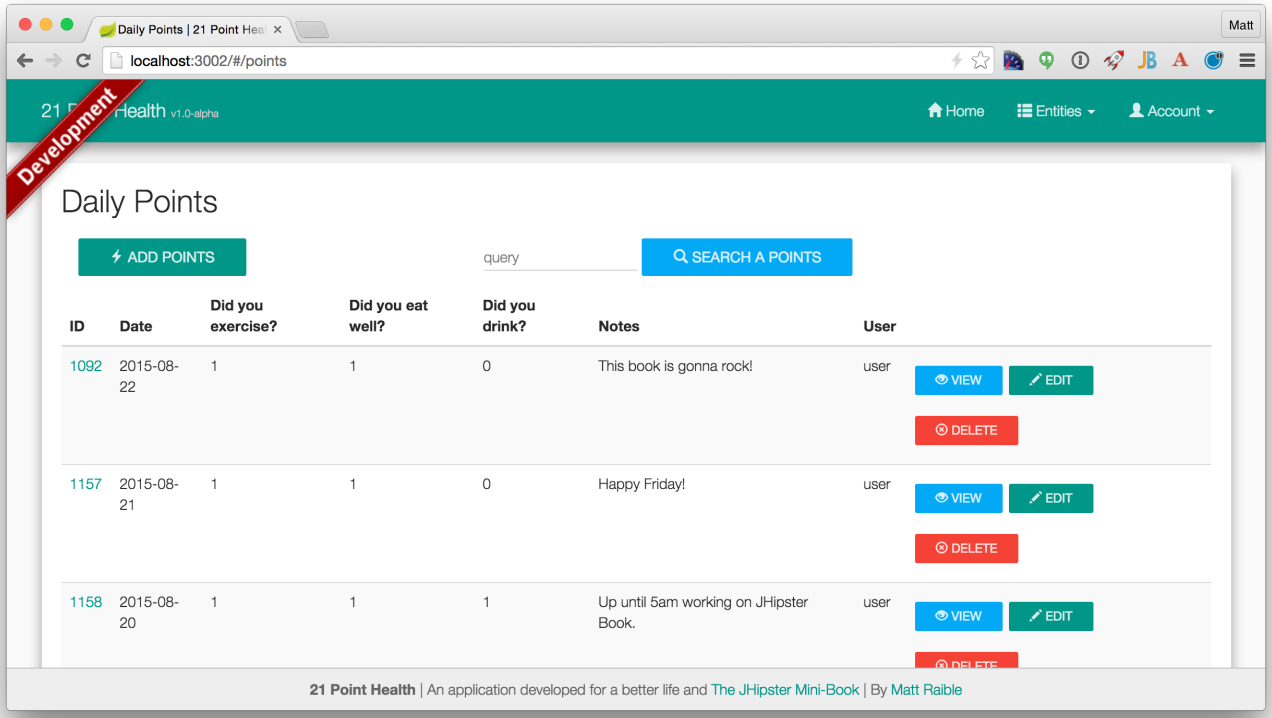


Figure 9. Default Daily Points list

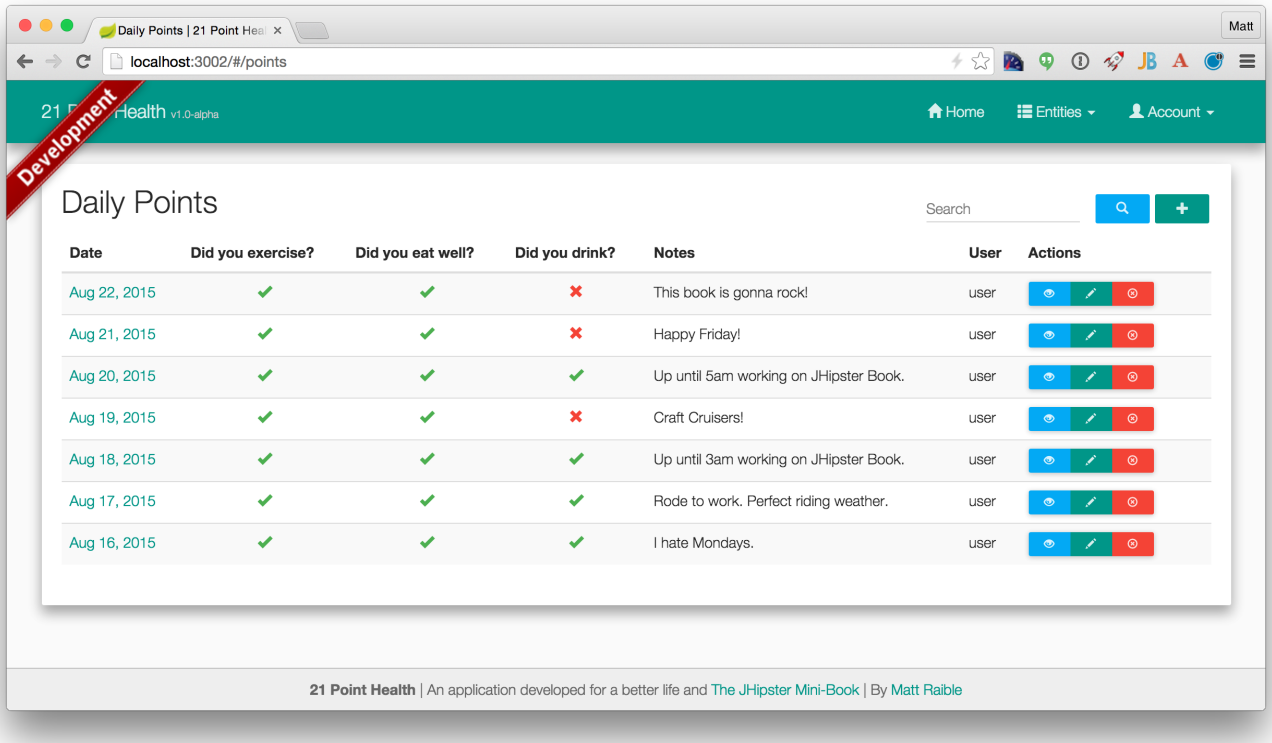


Figure 10. Default Daily Points list after UI improvements

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 a 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/scripts/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">
        <form name="searchForm" class="form-inline">
            <div class="form-group p-r">
                <input type="text" id="searchQuery"
                       class="form-control" ng-model="searchQuery"
                       placeholder="{{'entity.action.search' | translate}}">
            </div>
            <button class="btn btn-info btn-sm" ng-click="search()"
                    tooltip="{{'entity.action.search' | translate}}">
                <i class="glyphicon glyphicon-search"></i>
            </button>
            <button class="btn btn-primary btn-sm" ui-sref="points.new"
                    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/scripts/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/scripts/app/entities/points/points-dialog.html

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

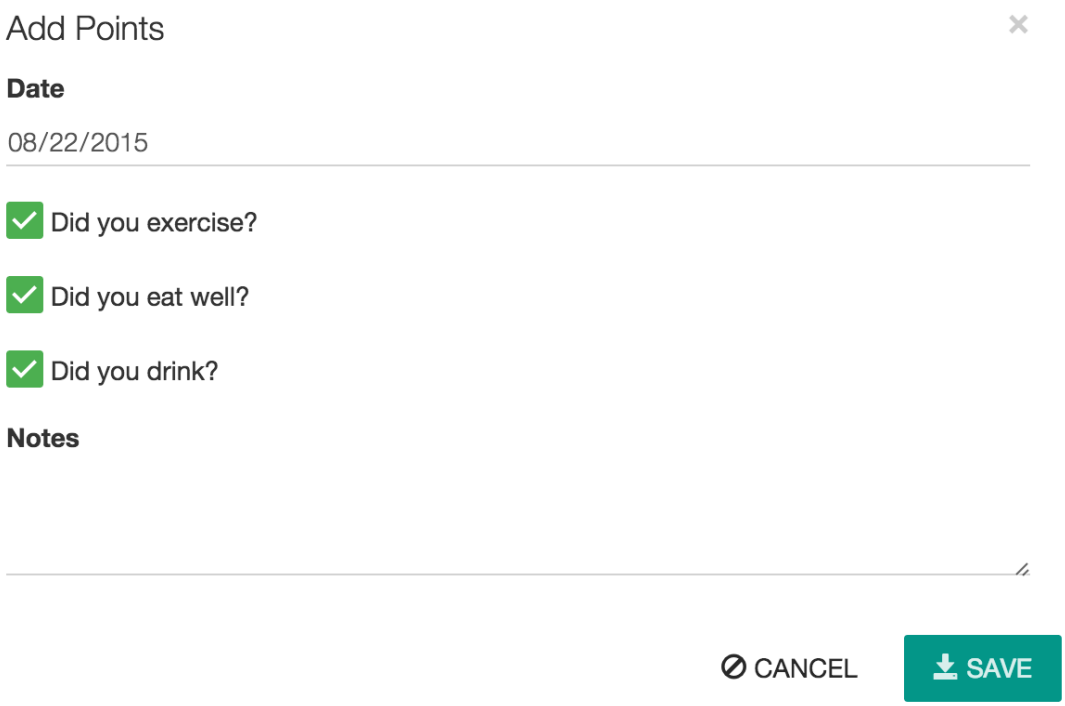


Figure 11. Add Points dialog

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

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

```
<div class="form-group" has-role="ROLE_ADMIN">
    <label translate="21pointsApp.weight.user" for="field_user">user</label>
    <select class="form-control" id="field_user" name="user" ng-model="weight.user.id"
            ng-options="user.id as user.login for user in users">
    </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.
      */
     @RequestMapping(value = "/points",
        method = RequestMethod.POST,
        produces = MediaType.APPLICATION_JSON_VALUE)
     @Timed
     public ResponseEntity<Points> create(@Valid @RequestBody Points points) throws
URISyntaxException {
         log.debug("REST request to save Points : {}", points);
         if (points.getId() != null) {
             return ResponseEntity.badRequest().header("Failure", "A new points cannot
already have an ID").body(null);
         }
+        if (!SecurityUtils.isUserInRole(AuthoritiesConstants.ADMIN)) {
+            log.debug("No user passed in, using current user: {}",
SecurityUtils.getCurrentLogin());
+
points.setUser(userRepository.findOneByLogin(SecurityUtils.getCurrentLogin()).get());
+        }
         Points result = pointsRepository.save(points);
```

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

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

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

To add Spring Security Test to the 21-Points Health project, I added `spring-security-test` to my `build.gradle`.

```
testCompile group: 'org.springframework.security', name: 'spring-security-test', version:
spring_security_version
```

I then modified `PointsResourceTest.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/PointsResourceTest.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;

@@ -63,18 +67,25 @@
     private PointsSearchRepository pointsSearchRepository;

     @Inject
+    private UserRepository userRepository;
+
+    @Inject
     private MappingJackson2HttpMessageConverter jacksonMessageConverter;

     private MockMvc restPointsMockMvc;

     private Points points;

+    @Autowired
+    private WebApplicationContext context;
+
     @PostConstruct
     public void setup() {
         MockitoAnnotations.initMocks(this);
         PointsResource pointsResource = new PointsResource();
         ReflectionTestUtils.setField(pointsResource, "pointsRepository",
pointsRepository);
         ReflectionTestUtils.setField(pointsResource, "pointsSearchRepository",
pointsSearchRepository);
+        ReflectionTestUtils.setField(pointsResource, "userRepository", userRepository);
```

```
         this.restPointsMockMvc =
MockMvcBuilders.standaloneSetup(pointsResource).setMessageConverters(jacksonMessageConver
ter).build();
     }

@@ -93,9 +104,15 @@
     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>> getAll(@RequestParam(value = "page", required =
false) Integer offset,
                                           @RequestParam(value = "per_page", required =
false) Integer limit)
    throws URISyntaxException {
    Page<Points> page;
    if (SecurityUtils.isUserInRole(AuthoritiesConstants.ADMIN)) {
        page = pointsRepository.findAll(PaginationUtil.generatePageRequest(offset, limit
));
    } else {
        page = pointsRepository.findAllForCurrentUser(PaginationUtil.generatePageRequest
(offset, limit));
    }
    HttpHeaders headers = PaginationUtil.generatePaginationHttpHeaders(page, "
/api/points", offset, limit);
    return new ResponseEntity<>(page.getContent(), headers, HttpStatus.OK);
}
```

The `PointsRepository#findAllForCurrentUser()` method that JHipster generated contains a custom query that uses Spring Expression Language to grab the user's information from Spring Security.

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

```
@Query("select points from Points points where points.user.login =
?#{principal.username}")
Page<Points> findAllForCurrentUser(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 `findAll` to `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 `PointsResourceTest#getAllPoints` to use Spring Security Test's `user` post processor.

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

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

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

     // Get all the points
-    restPointsMockMvc.perform(get("/api/points"))
+    restPointsMockMvc.perform(get("/api/points")
+            .with(user("admin").roles("ADMIN")))
             .andExpect(status().isOk())
```

Implementing the UI mockup

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

1. Adding buttons to facilitate adding new data from the homepage.
2. Adding an API to get points achieved during the current week.
3. Adding an API to get blood-pressure readings for the last 30 days.
4. Adding an API to get body weights for the last 30 days.
5. Adding 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 `main.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 `main` state. I copied the generated `points.new` state from `points.js` and pasted it into `main.js`.

src/main/webapp/scripts/app/main/main.js

```
.state('points.add', { ①
    parent: 'home', ②
    url: 'add/points', ③
    data: {
        roles: ['ROLE_USER']
    },
    onEnter: ['$stateParams', '$state', '$modal', function($stateParams, $state, $modal)
{
        $modal.open({
            templateUrl: 'scripts/app/entities/points/points-dialog.html',
            controller: 'PointsDialogController',
            size: 'lg',
            resolve: {
                entity: function () {
                    return {date: null, exercise: null, meals: null, alcohol: null,
notes: null, id: null};
                }
            }
        }).result.then(function(result) { 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/scripts/app/main/main.html

```
<div class="col-md-4 text-right">
    <a ui-sref="points.add" class="btn btn-primary btn-raised">Add Points</a>
</div>
```

Points this week

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

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

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

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

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

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

@Test
@Transactional
public void getPointsThisWeek() throws Exception {
    LocalDate today = new LocalDate();
    LocalDate thisMonday = today.withDayOfWeek(DateTimeConstants.MONDAY);
    LocalDate lastMonday = thisMonday.minusWeeks(1);
    createPointsByWeek(thisMonday, lastMonday);

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

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

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

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

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

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

```
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;
    }

    @JsonSerialize(using = CustomLocalDateSerializer.class)
    @JsonDeserialize(using = ISO8601LocalDateDeserializer.class)
    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 = new LocalDate(); ①
    // Get first day of week
    LocalDate startOfWeek = now.withDayOfWeek(DateTimeConstants.MONDAY); ②
    // Get last day of week
    LocalDate endOfWeek = now.withDayOfWeek(DateTimeConstants.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.getCurrentLogin()))
        .mapToInt(p -> p.getExercise() + p.getMeals() + p.getAlcohol())
        .sum();

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

① I later discovered that creating a new `LocalDate` uses the server's time zone by default. When I deployed on a server using UTC, I discovered this logic didn't work too well. I decided I'd make it a user preference or look into using a JavaScript library like jsTimezoneDetect to detect a client's time zone and pass it to the server.

② Since I live in the United States, I'm used to the week beginning on Sunday. However, since Joda-Time uses Monday as the first day of the week, I decided this would be my application's logic as well.

To support this new method on the client, I added a new method to my `Points` service.

src/main/webapp/scripts/components/entities/points/points.service.js

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

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

src/main/webapp/scripts/app/main/main.controller.js

```
.controller('MainController', function ($scope, Principal, Points) {
    Principal.identity().then(function(account) {
        $scope.account = account;
        $scope.isAuthenticated = Principal.isAuthenticated;
    });

    Points.thisWeek(function(data) {
        $scope.pointsThisWeek = data;
        $scope.pointsPercentage = (data.points / 21) * 100;
    });
});
```

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

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

```
<div class="row">
    <div class="col-md-10">
        <div class="progress progress-lg" ng-show="pointsThisWeek.points"> ①
            <div class="progress-bar progress-bar-success progress-bar-striped" role=
"progressbar"
                 aria-valuenow="{{pointsThisWeek.points}}"
                 aria-valuemin="0" aria-valuemax="21" style="width: {{pointsPercentage}}
%">
                 {{pointsThisWeek.points}} / Goal: 10
            </div>
        </div>
        <alert type="info" ng-hide="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 entering some data for the current user.

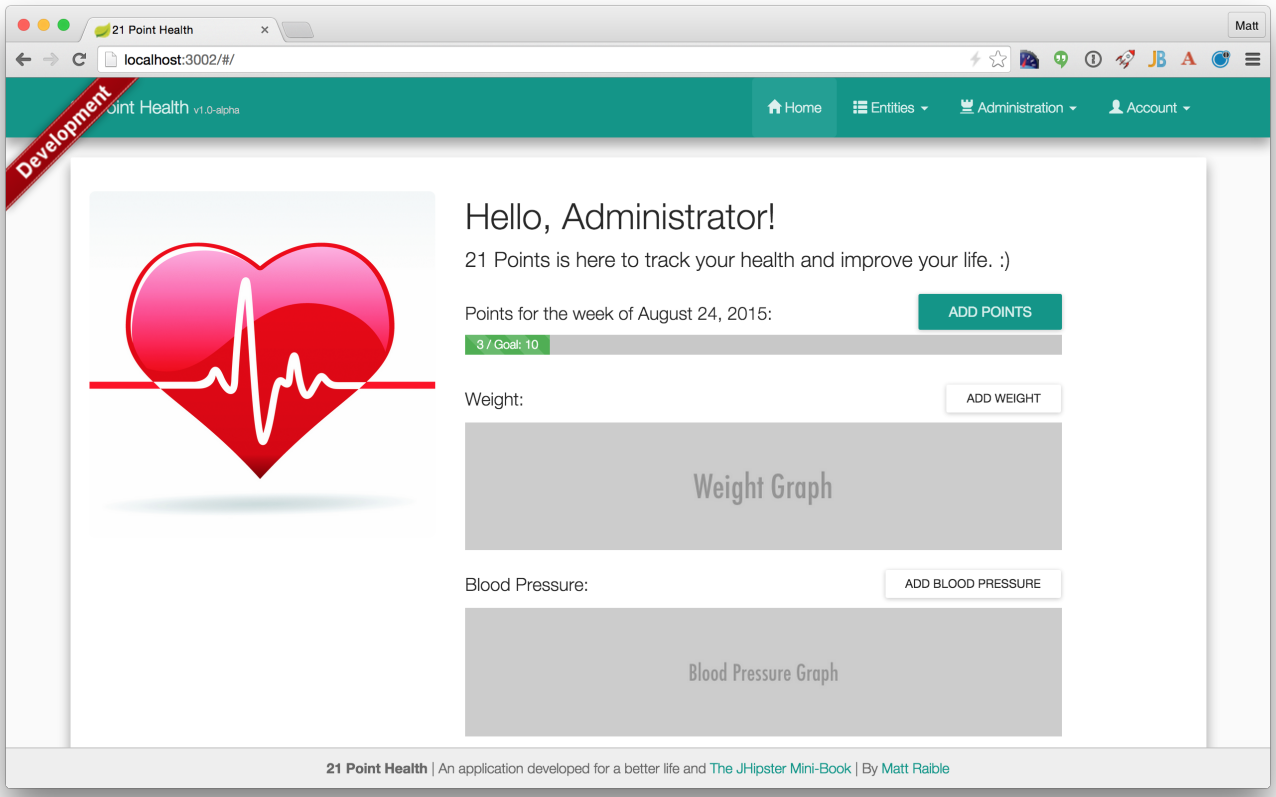


Figure 12. Progress bar for points this week

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. 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")
@Timed
public ResponseEntity<Preferences> getUserPreferences() {
    String username = SecurityUtils.getCurrentLogin();
    log.debug("REST request to get Preferences : {}", username);
    User user = userRepository.findOneByLogin(username).get();

    if (user.getPreferences() != null) {
        return new ResponseEntity<>(user.getPreferences(), 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/scripts/components/entities/preferences/preferences.service.js

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

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

src/main/webapp/scripts/app/main/main.controller.js

```
.controller('MainController', function ($scope, Principal, Points, Preferences) {
    ...

    Preferences.user(function(data) {
        $scope.preferences = data;
    })
});
```

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

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

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

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

```
private void createBloodPressureByMonth(DateTime firstOfMonth, DateTime
firstDayOfLastMonth) {
    User user = userRepository.findOneByLogin("user").get();
    // this month
    bloodPressure = new BloodPressure(firstOfMonth, 120, 80, user);
    bloodPressureRepository.saveAndFlush(bloodPressure);
    bloodPressure = new BloodPressure(firstOfMonth.plusDays(10), 125, 75, user);
    bloodPressureRepository.saveAndFlush(bloodPressure);
    bloodPressure = new BloodPressure(firstOfMonth.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 {
    DateTime now = new DateTime();
    DateTime firstOfMonth = now.withDayOfMonth(1);
    DateTime firstDayOfLastMonth = firstOfMonth.minusMonths(1);
    createBloodPressureByMonth(firstOfMonth, firstDayOfLastMonth);

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

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

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

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

src/main/java/org/jhipster/health/web/rest/dto/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(DateTime firstDate,
DateTime 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) {
    LocalDate today = new LocalDate();
    LocalDate previousDate = today.minusDays(days);
    DateTime daysAgo = previousDate.toDateTimeAtCurrentTime();
    DateTime rightNow = today.toDateTimeAtCurrentTime();

    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());
}
```

I added a new method to support this API in `bloodPressure.service.js`.

src/main/webapp/scripts/components/entities/bloodPressure/bloodPressure.service.js

```
.factory('BloodPressure', function ($resource, DateUtils) {
    return $resource('api/bloodPressures/:id', {}, {
        '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 `grunt wiredep` to update `index.html` and `karma.conf.js` with references to the new files. I also updated `app.js` to add `nvd3` as a dependency.

src/main/webapp/scripts/app/app.js

```
angular.module('21pointsApp', ['LocalStorageModule', 'tmh.dynamicLocale',
'pascalprecht.translate',
    'ui.bootstrap', // for modal dialogs
    'ngResource', 'ui.router', 'ngCookies', 'ngCacheBuster', 'ngFileUpload', 'infinite-
scroll', 'nvd3'])
```

I modified `main.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.

src/main/webapp/scripts/component/chart/chart.service.js

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

src/main/webapp/scripts/app/main/main.controller.js

```
BloodPressure.last30Days(function(bpReadings) {
    $scope.bpReadings = bpReadings;
    if (bpReadings.readings.length) {
        $scope.bpOptions = angular.copy(Chart.getBpChartConfig());
        $scope.bpOptions.title.text = bpReadings.period;
        $scope.bpOptions.chart.yAxis.axisLabel = "Blood Pressure";
        var systolics, diastolics;
        systolics = [];
        diastolics = [];
        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
            });
        });
        $scope.bpData = [{
            values: systolics,
            key: 'Systolic',
            color: '#673ab7'
        }, {
            values: diastolics,
            key: 'Diastolic',
            color: '#03a9f4'
        }];
    }
});
```

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

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

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

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

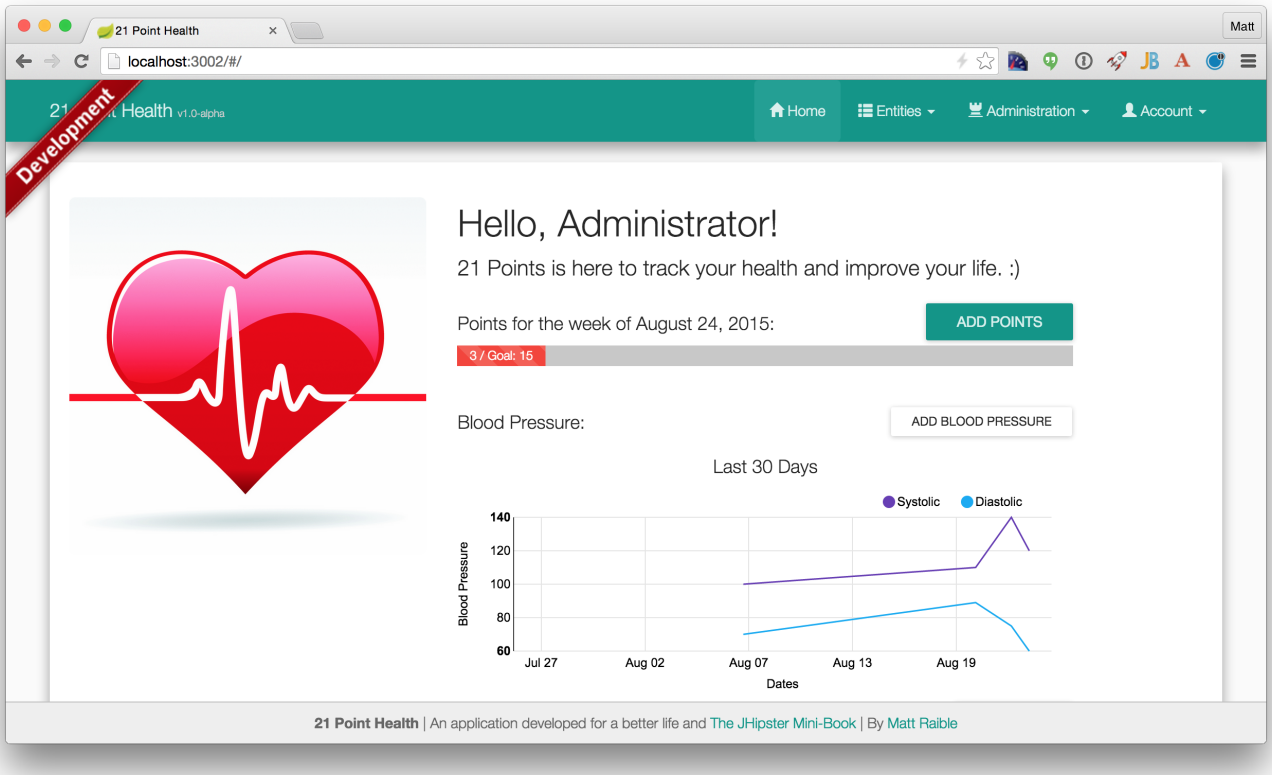


Figure 13. Chart of blood pressure during the last 30 days

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 1,152 lines of code. JHipster did the rest for me, generating 91.7% of the code in my project!

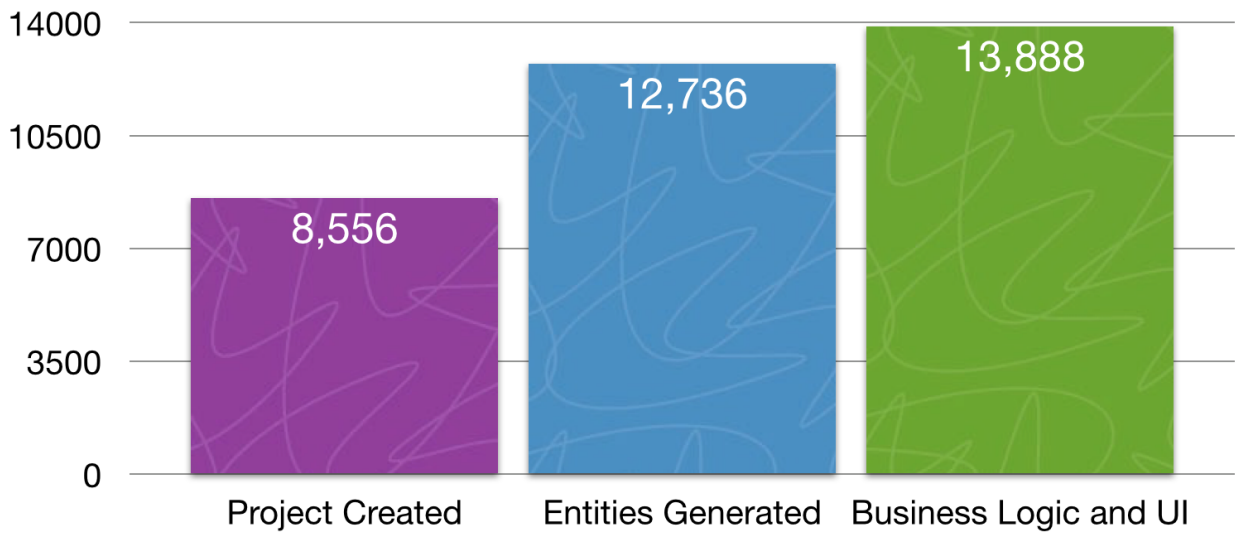


Figure 14. Project lines of code

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

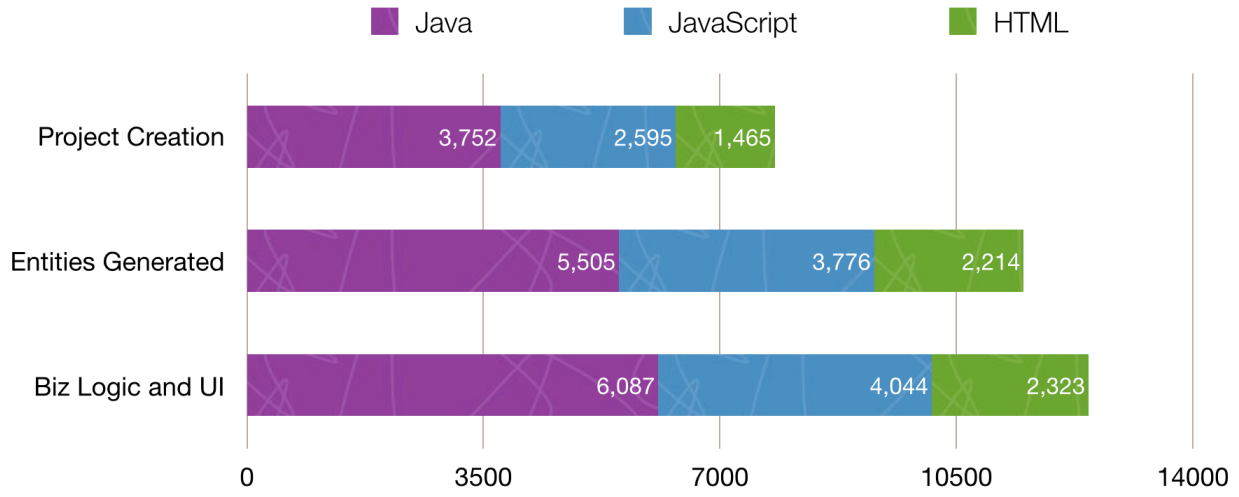


Figure 15. Project lines of code by language

The amount of code I had to write in each language was 582 lines of Java, 268 lines of JavaScript, and 109 lines of HTML.

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 Joda-Time 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, OpenShift, and AWS. 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 Grunt or Gulp by running `grunt build` or `gulp build`, depending on which tool your project uses. 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 Toolbelt 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
Installing https://github.com/heroku/heroku-deploy...
done

Creating Heroku application and setting up node environment
heroku create 21points --addons heroku-postgresql:hobby-dev
   { [Error: Command failed: /bin/sh -c heroku create 21points --addons heroku-
postgresql:hobby-dev
 !    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 21points --addons heroku-postgresql:hobby-dev' }
```

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
Installing https://github.com/heroku/heroku-deploy...
done

Creating Heroku application and setting up node environment
heroku create health-by-points --addons heroku-postgresql:hobby-dev
Creating health-by-points... done, stack is cedar-14

Adding heroku-postgresql:hobby-dev to health-by-points...
done

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

Git remote heroku added

Creating Heroku deployment files

Building application
:generateMainMapperClasses

Download
https://oss.sonatype.org/content/repositories/releases/io/dropwizard/metrics/metrics-
healthchecks/3.1.2/metrics-healthchecks-3.1.2.pom
...

BUILD SUCCESSFUL

Total time: 2 mins 58.204 secs

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

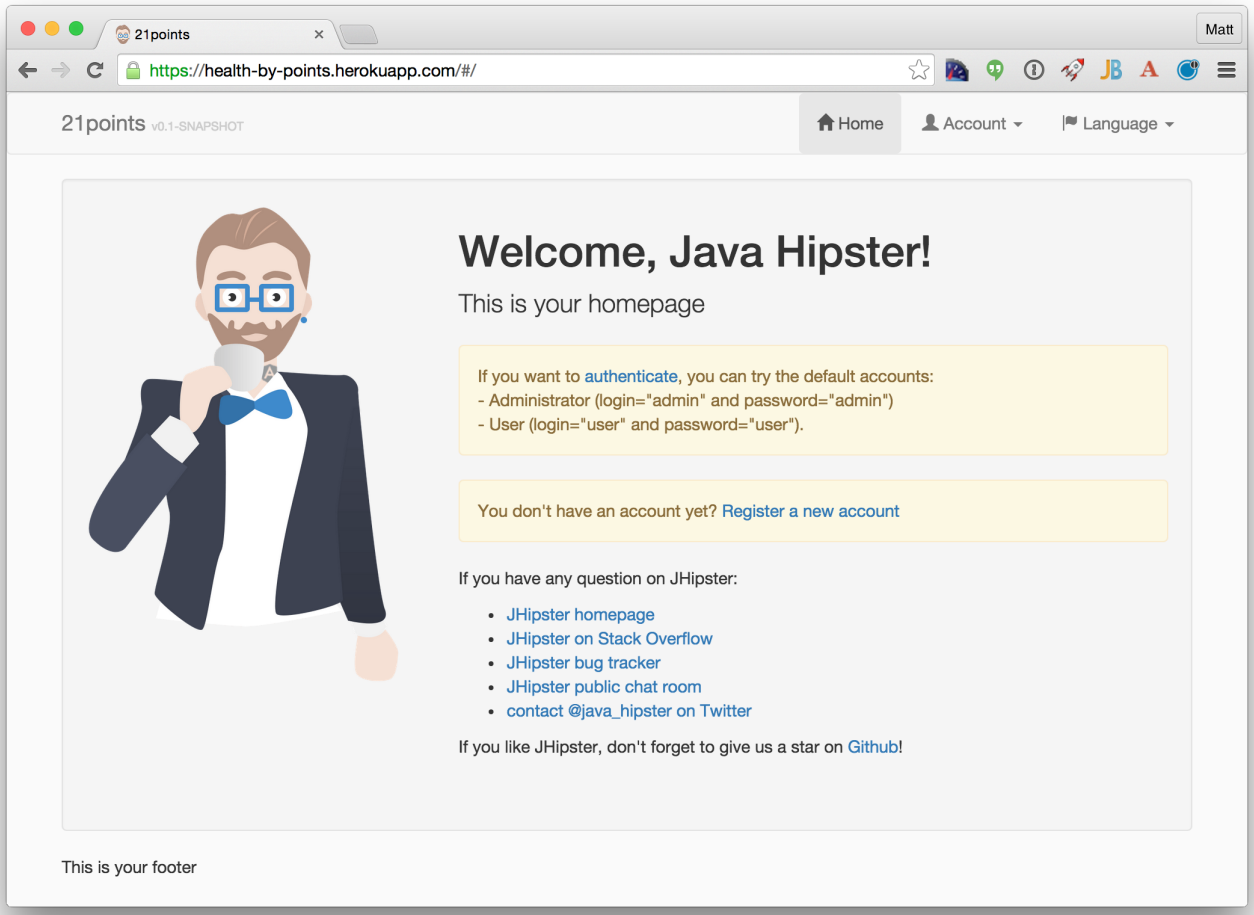


Figure 16. First deployment to Heroku

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 my 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, 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 and deploying to Heroku. The CI documentation also shows how to set up Jenkins on Linux and Windows. However, it doesn't have any documentation on how to configure a job to build, test, and deploy. This section aims to supply that.

I added `jasmine-reporters` and `karma-junit-reporter` to my project so I could read JavaScript test results with Jenkins.

```
npm install jasmine-reporters --save-dev
npm install karma-junit-reporter --save-dev
```

Then I updated `src/test/javascript/karma.conf.js` to override the default plugins, specify reporters, and configure the JUnit reporter.

src/test/javascript/karma.conf.js

```
singleRun: false,

plugins: [
    'karma-chrome-launcher',
    'karma-phantomjs-launcher',
    'karma-jasmine',
    'karma-junit-reporter'
],

reporters: ['dots', 'junit'],

junitReporter: {
    outputFile: '../build/test-results/TEST-javascript-results.xml',
    suite: 'unit'
}
```

Testing in the Zone

I created a new Karma configuration in `Gruntfile.js` to allow continuous testing while "in the zone". Running `grunt karma:zone` will automatically re-execute tests when you save changes.

Gruntfile.js

```
karma: {
    unit: {
        configFile: 'src/test/javascript/karma.conf.js',
        singleRun: true
    },
    zone: {
        configFile: 'src/test/javascript/karma.conf.js',
        singleRun: false,
        autoWatch: true
    }
}
```

To make this work, I also had to update the `test` task to run `karma:unit` instead of `karma`.

I set up a new freestyle project in Jenkins using this configuration.

- Project name: `21-points`
- Source Code Management
  - Git Repository: `git@bitbucket.org:mraible/21-points.git`

    - Branches to build: `*/master`
    - Additional Behaviours: `Wipe out repository & force clone`
- Build Triggers
    - Poll SCM / Schedule: `H/5 * * * *`
- Build
    - Invoke Gradle script / Use Gradle Wrapper / Tasks: `-Pprod clean test bootRepackage`
- Post-build Actions
    - Build other projects: `21-points-deploy`
    - Publish JUnit test result report / Test Report XMLs: `build/test-results/*.xml`

I then created another job to deploy to Heroku.

- Project name: `21-points-deploy`
- Source Code Management
    - Git Repository: `git@bitbucket.org:mraible/21-points.git`
    - Branches to build: `*/master`
- Build
    - Invoke Gradle script / Use Gradle Wrapper / Tasks: `-Pprod bootRepackage -x test`
    - Execute Shell / Command: `heroku deploy:jar --jar build/libs/*.war --app health-by-points`

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.

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`. It showed you how to modify relationships between entities after the fact and how Liquibase can generate changelogs to update your database. 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,

JavaScript build tools, 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.

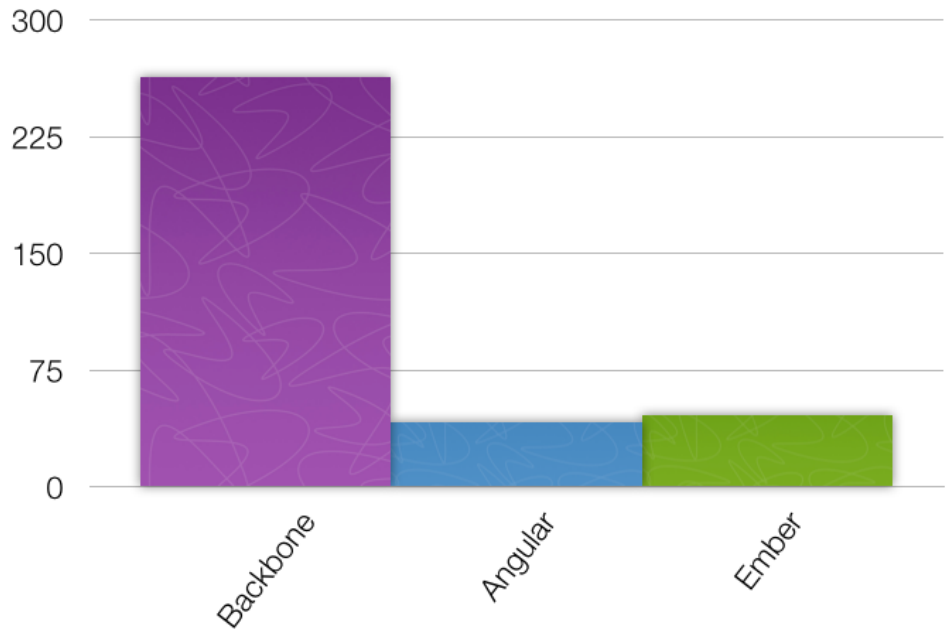


Figure 17. Jobs on Dice, March 2013

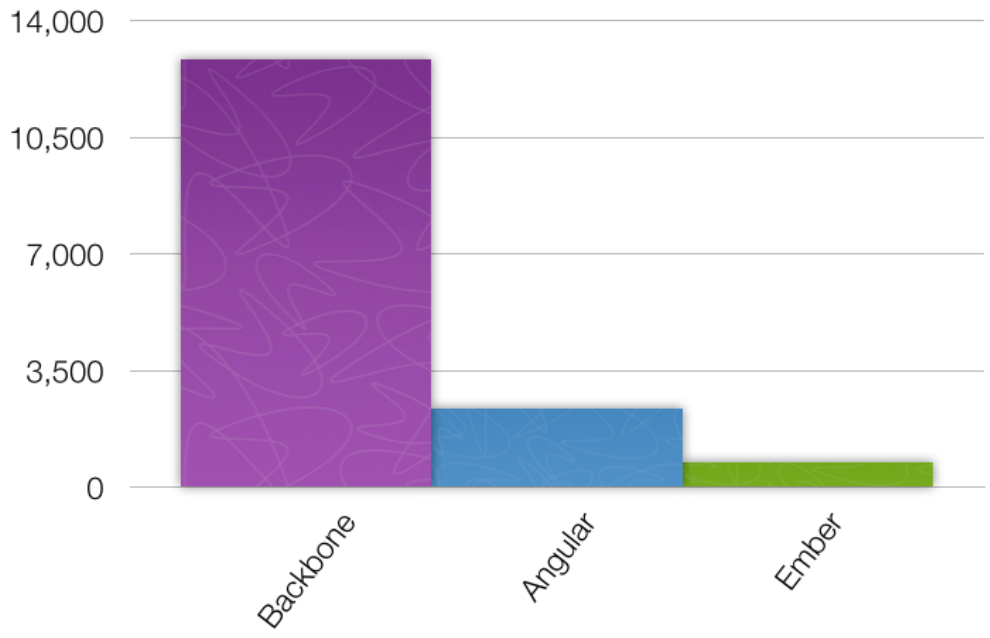


Figure 18. Skills on LinkedIn, March 2013

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%!

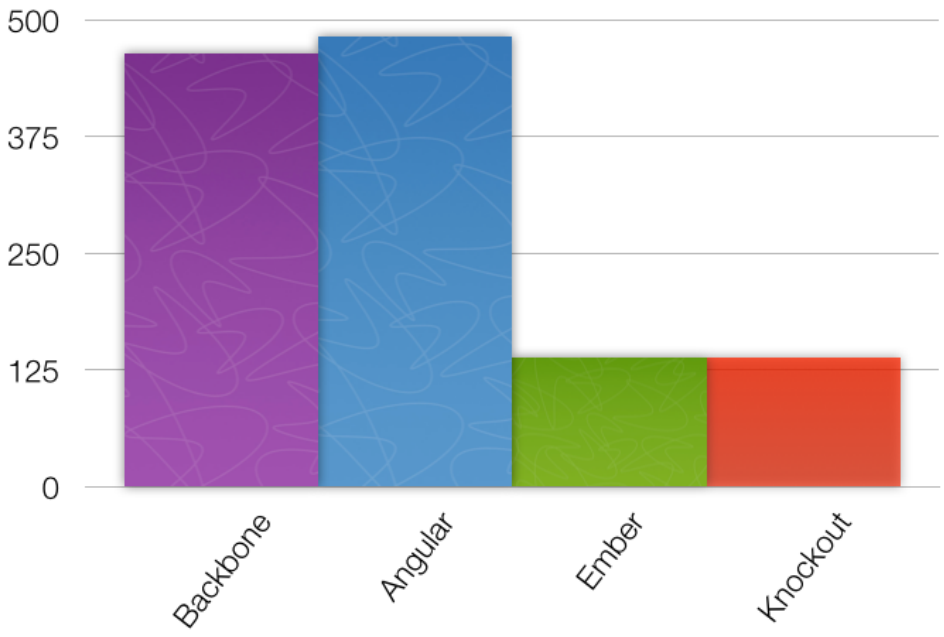


Figure 19. Jobs on Dice, February 2014

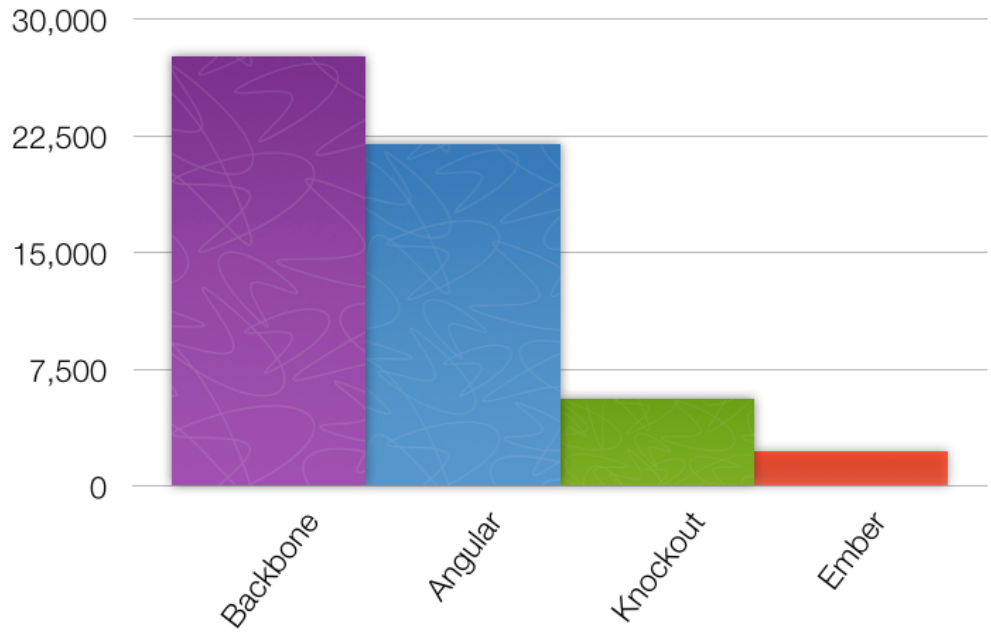


Figure 20. Skills on LinkedIn, February 2014

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 seperation 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/1.3.11/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.

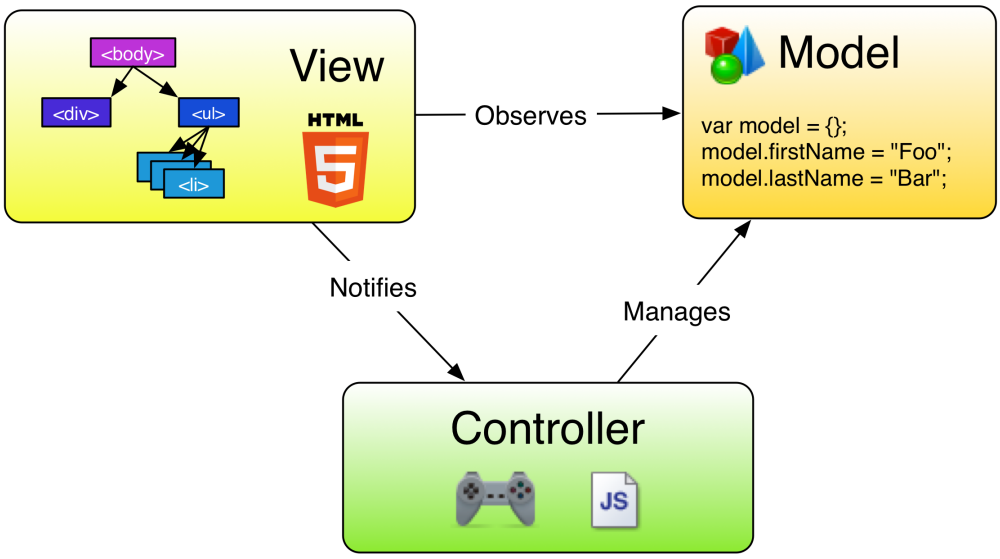


Figure 21. MVC in Angular

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 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.

```
.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/1.3.11/angular.min.js"></script>
<script src="http://code.angularjs.org/1.3.11/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 and 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/1.3.11/angular.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-router/0.2.14/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 *voila*, 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 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).

AngularJS 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 AngularJS 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 object
- `$scope`
- `angular.module`
- 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.

JHipster will add support for Angular 2 once it's deemed mature enough for production use. In the meantime, I recommend you read and follow John Papa's Angular Style Guide to prepare for Angular 2.

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 and 1170 pixels, according to your viewport. On viewports narrower than 767 pixels, the columns become fluid and stack vertically.

A basic example of the grid:

```
<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 pixels.

Figure 22. Basic grid on desktop

If you squish your browser to less than 1200 pixels 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>
```

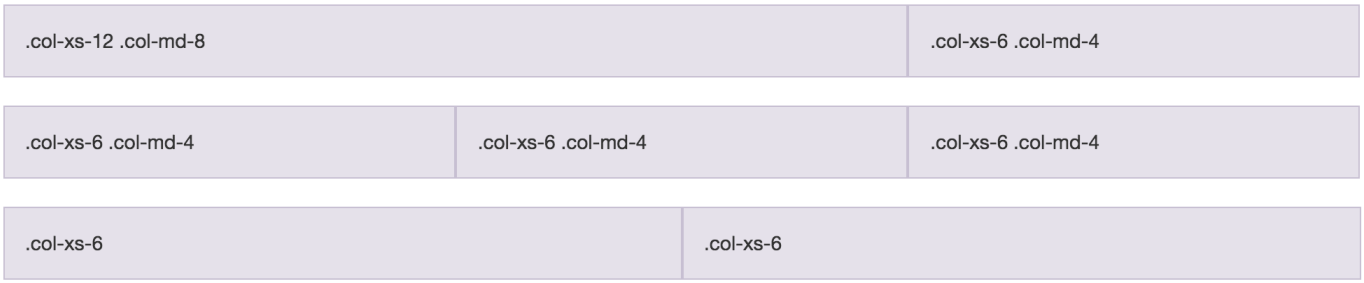


Figure 24. Advanced grid

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. Boostrap 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.

- `.visible-[xs|sm|md|lg]`
- `.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>
```

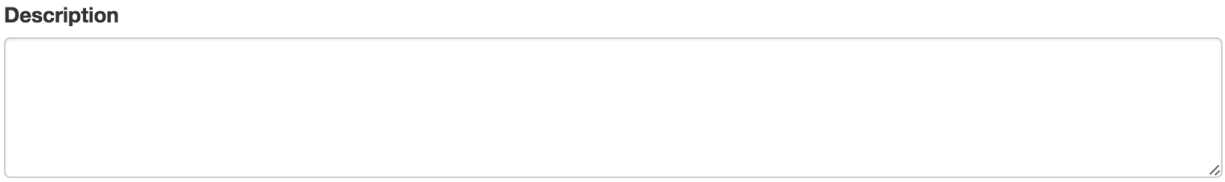


Figure 25. Basic form element

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>
```

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.

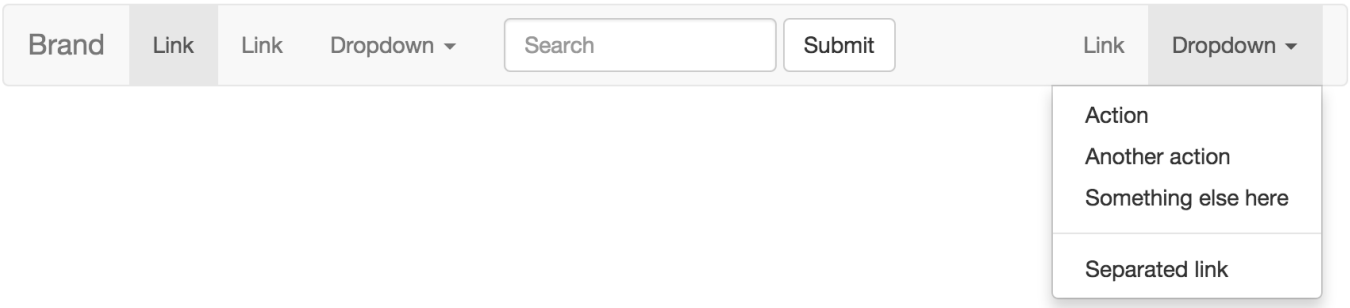


Figure 28. Navbar with dropdown

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

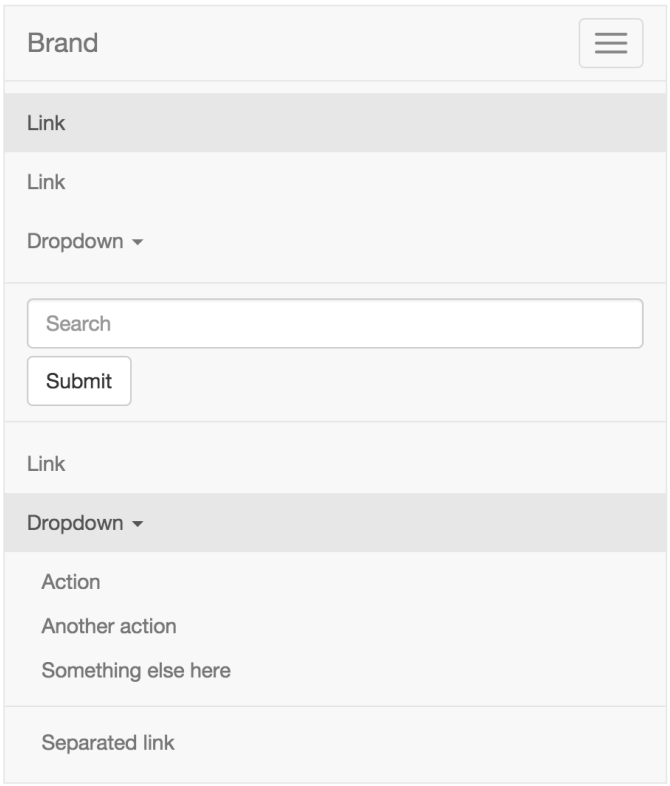


Figure 29. Navbar on mobile

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.

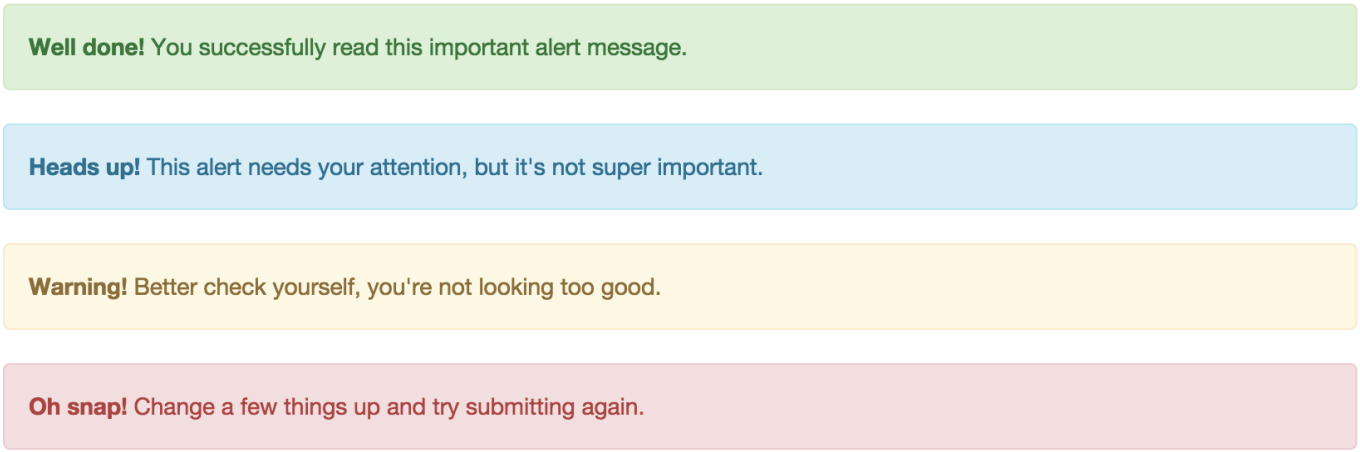


Figure 30. Alerts

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>
```

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.

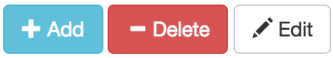


Figure 32. Buttons with icons

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/assets/styles/main.css` directly. If you're using Compass to compile `*.scss` files, you can modify `src/main/scss/main.scss` instead. Using Compass results in a much more concise authoring environment. Below is the default `main.scss` file that Yeoman generates. You can see that it overrides the font location variable, imports Bootstrap, then overrides some of its default `body` rules.

src/main/scss/main.scss

```
$icon-font-path: "../../bower_components/bootstrap-sass/assets/fonts/bootstrap/";

@import "bootstrap-sass/assets/stylesheets/_bootstrap.scss";

body {
    background: #fafafa;
    font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
    color: #333;
}
```

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 "translate" attribute with a key.

```
<label 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 installs English and French translations when you first create a project. It stores the JSON documents for these languages in `src/main/webapp/i18n`, in `en` and `fr` directories. You can install additional languages using `yo jhipster:languages`. At the time of this writing, JHipster supports 14 additional languages. You can also add a new language. To set the default language, modify `src/main/webapp/scripts/app/app.js` and its preferred language line.

src/main/webapp/scripts/app/app.js

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

Sass

Sass stands for "Syntactically Awesome StyleSheets". 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 Compass. To learn more about structuring your CSS and naming classes, read the great *Scalable and Modular Architecture for CSS*.

Grunt versus Gulp

JHipster prompts you to choose between Grunt and Gulp for building the client. Both tools allow you to perform tasks like minification, concatenation, compilation (e.g. from TypeScript/ CoffeeScript to JavaScript), unit testing, and more.

Grunt

Grunt calls itself "the JavaScript task runner". It's the incumbent in the field of JavaScript build tools and has over 4,000 plugins. You install and manage Grunt and Grunt 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 Grunt globally (so you can use it anywhere), run `npm install -g grunt-cli`.

Grunt was created by Ben Alman, who first released it March 29, 2012.

The following steps show how to integrate Grunt 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 Grunt and its plugins for web optimization.

package.json

```
{
  "name": "grunt-example",
  "description": "A Grunt examplek",
  "version": "1.0.0",
  "devDependencies": {
    "grunt": "~0.4.5",
    "grunt-contrib-concat": "~0.5.1",
    "grunt-contrib-uglify": "~0.9.1",
    "grunt-contrib-cssmin": "~0.12.2",
    "grunt-usemin": "~3.0.0",
    "grunt-contrib-copy": "~0.8.0",
    "grunt-rev": "~0.1.0",
    "grunt-contrib-clean": "~0.6.0",
    "load-grunt-tasks": "3.1.0"
  }
}
```

Then, you have to write a `Gruntfile.js` that configures each of the different tasks for the project.

Gruntfile.js

```
module.exports = function (grunt) {
    require('load-grunt-tasks')(grunt);

    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),

        clean: ["dist", '.tmp'],

        copy: {
            main: {
                expand: true, cwd: 'app/',
                src: ['**', '!js/**', '!lib/**', '!**/*.css'], dest: 'dist/'
            }
        },

        rev: {files: {src: ['dist/**/*.{js,css}']}},

        useminPrepare: {html: 'app/index.html'},

        usemin: {html: ['dist/index.html']},

        uglify: {options: {report: 'min', mangle: false}}
    });

    // Tell Grunt what to do when we type "grunt" into the terminal
    grunt.registerTask('default', ['copy', 'useminPrepare', 'concat', 'uglify', 'cssmin',
'rev', '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/f050d0dc.f050d0dc.seed.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/bd6abc62.bd6abc62.seed.min.js"></script>
</body>
</html>
```

Gulp

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 compare Gulp versus Grunt, I integrated it into the angular-seed project. Below are the `package.json` and `gulpfile.js` that optimize everything for production. The input and resulting HTML are the same as they were for the Grunt example.

package.json

```
{
  "name": "gulp-example",
  "description": "A Gulp example",
  "version": "1.0.0",
  "devDependencies": {
    "gulp": "~3.8.11",
    "gulp-clean": "~0.3.1",
    "gulp-concat": "~2.5.2",
    "gulp-load-plugins": "~0.10.0",
    "gulp-minify-css": "~1.1.1",
    "gulp-rev": "~3.0.1",
    "gulp-uglify": "~1.2.0",
    "gulp-usemin": "~0.3.11"
  }
}
```

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($.usemin({
            css: [$.minifyCss(), $.rev()],
            js: [$.uglify(), $.rev()]
        }))
        .pipe(gulp.dest('dist/'));
});

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

You can see from this example that Gulp is a bit more concise. However, both tools have their merits.

Gulp is the newcomer so it can be a bit more nimble without the need for backwards compatibility. It also has many fewer plugins (just over 1,500 at the time of this writing). If you compare Grunt and Gulp at Google Trends, you'll see that Gulp received a spike of hype when it first came out but that Grunt has usually led by a slight margin.

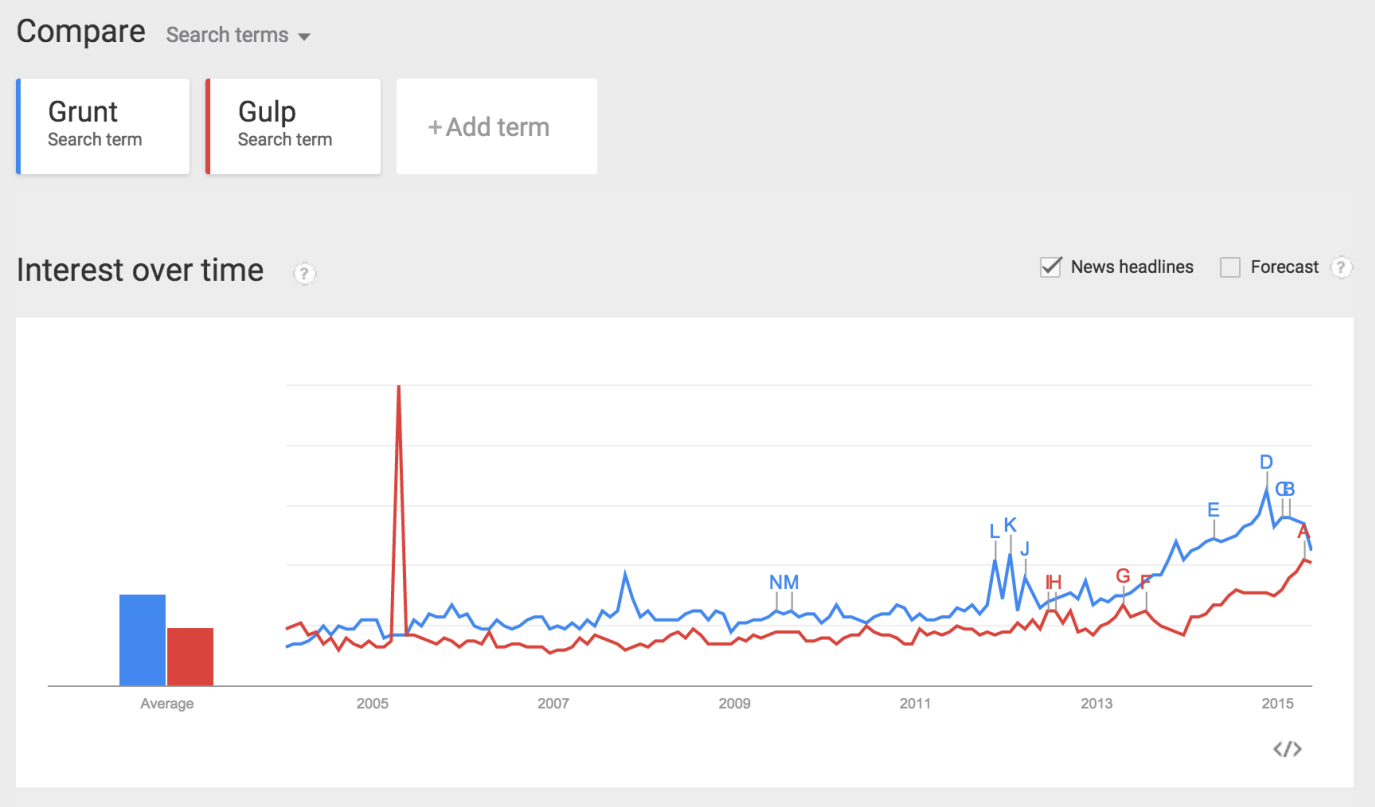


Figure 33. Google Trends: Grunt vs. Gulp

To learn more, see Mark Dalgleish'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 article. Since webpack is not an option for JHipster, I will not cover it in this book.

(Keith Cirkel recommends you use neither Grunt nor Gulp but use npm.)

If you're new to JavaScript build tools, I recommend Grunt. You don't want to be overwhelmed with too much new technology in your JHipster project if you're trying to meet a deadline. If you're using JHipster to learn about a bunch of new technologies, try Gulp, which seems to be the hipper option.

WebSockets

WebSockets is an advanced technology that makes it possible to open an interactive communication channel between the user's browser and a server. With this API, you can send messages to a server and receive event-driven responses without 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-orientated 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/scripts/app/admin/tracker/tracker.controller.js

```
angular.module('blogApp')
    .controller('TrackerController', function ($scope) {
        $scope.activities = [];
        var stompClient = null;
        var socket = new SockJS('/websocket/tracker');
        stompClient = Stomp.over(socket);
        stompClient.connect({}, function(frame) {
            stompClient.subscribe('/topic/activity', function(activity){
                showActivity(JSON.parse(activity.body));
            });
        });

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

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

```
'use strict';

angular.module('blogApp')
    .factory('Tracker', function ($rootScope) {
        var stompClient = null;

        function sendActivity() {
            stompClient.send('/websocket/activity', {},
                JSON.stringify({'page': $rootScope.toState.name}));
        }

        return {
            connect: function () {
                var socket = new SockJS('/websocket/activity');
                stompClient = Stomp.over(socket);
                stompClient.connect({}, function(frame) {
                    sendActivity();
                    $rootScope.$on('$stateChangeStart', function (event) {
                        sendActivity();
                    });
                });
            },
            sendActivity: function () {
                if (stompClient != null) {
                    sendActivity();
                }
            },
            disconnect: function() {
                if (stompClient != null) {
                    stompClient.disconnect();
                    stompClient == null;
                }
            }
        };
    });
```

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 `Gruntfile.js`. Your `package.json` file only needs to contain a few things, weighing in at a slim 15 lines of JSON.

```
{
  "name": "jhipster-book",
  "version": "1.0.0",
  "description": "The JHipster Mini-Book",
  "repository": {
    "type": "git",
    "url": "git@bitbucket.org:mraible/jhipster-book.git"
  },
  "devDependencies": {
    "grunt": "0.4.5",
    "grunt-browser-sync": "2.0.0",
    "load-grunt-tasks": "3.1.0",
    "grunt-contrib-watch": "0.6.1"
  }
}
```

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

```
'use strict';

module.exports = function (grunt) {
    require('load-grunt-tasks')(grunt);

    grunt.initConfig({
        watch: {
            web: {
                files: ['src/main/webapp/css/*.css', 'src/main/webapp/js/**/*.js']
            }
        },
        browserSync: {
            dev: {
                bsFiles: {
                    src : [
                        'src/main/webapp/index.html',
                        'src/main/webapp/tpl/**/*.html',
                        'src/main/webapp/css/*.css',
                        'src/main/webapp/js/**/*.js',
                        'src/main/webapp/img/**/*.{png,jpg,jpeg,gif,webp,svg}'
                    ]
                }
            },
            options: {
                watchTask: true,
                server: {
                    baseDir: "./src/main/webapp/"
                }
            }
        }
    });

    grunt.registerTask('serve', [
        'browserSync',
        'watch'
    ]);

    grunt.registerTask('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 Grunt. 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.

```
grunt serve
```

For an audiovisual introduction to Browsersync, check out the introduction video on its homepage.

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 Grunt and 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 compliments 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.

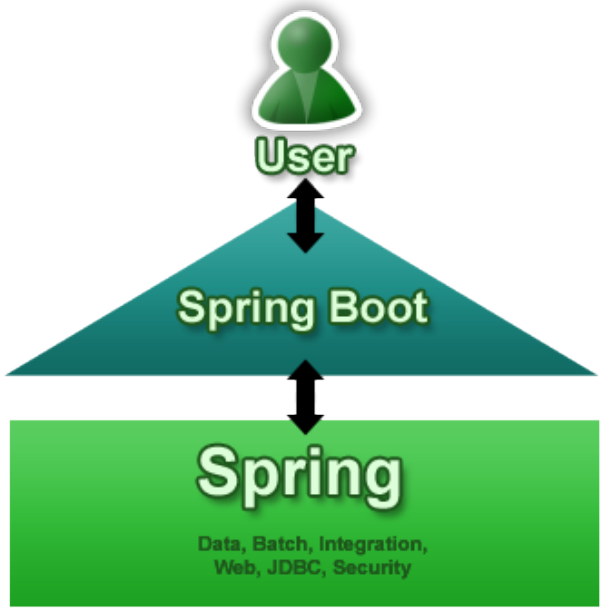


Figure 34. Spring Boot

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.
- 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`.

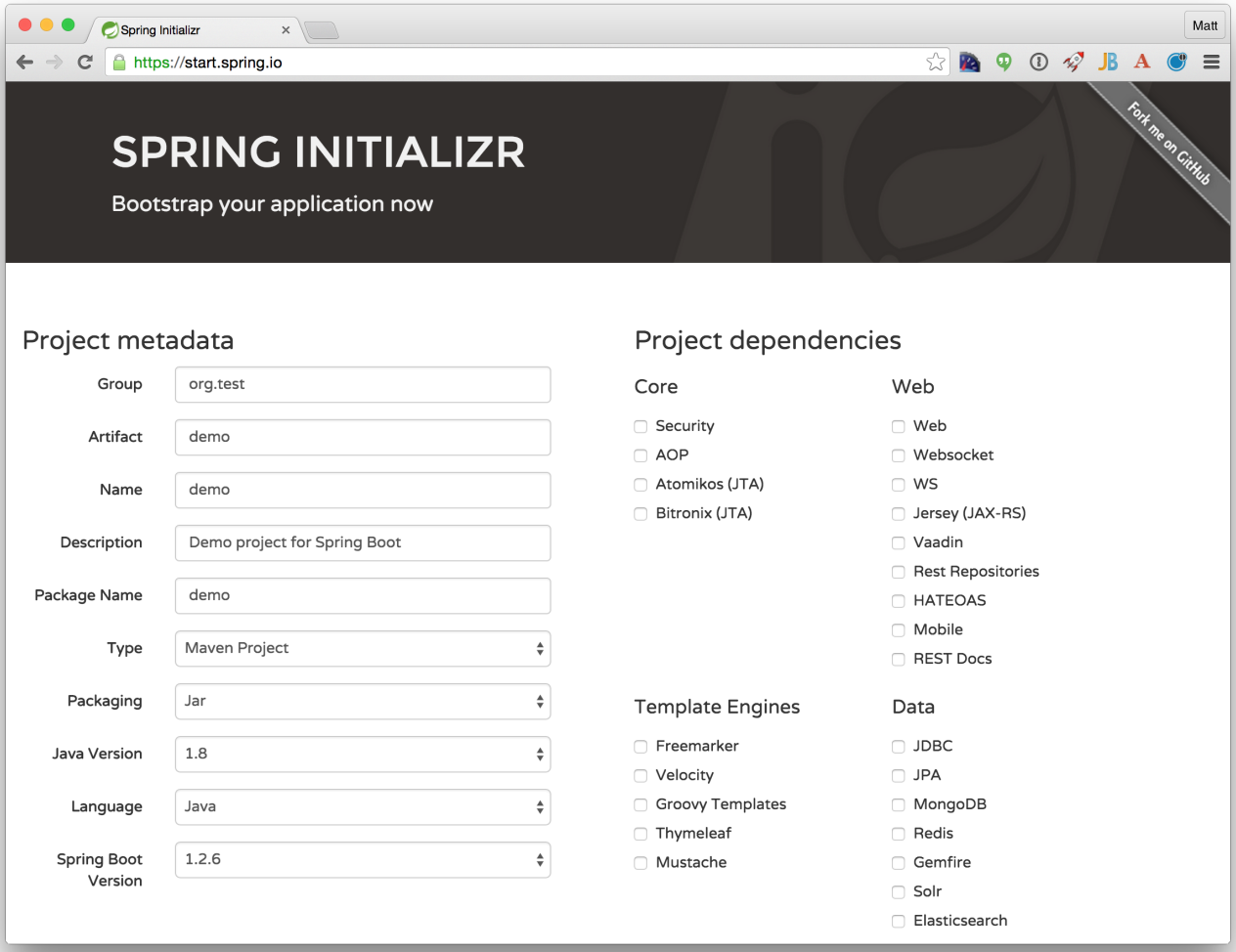


Figure 35. Spring Initializr in a browser

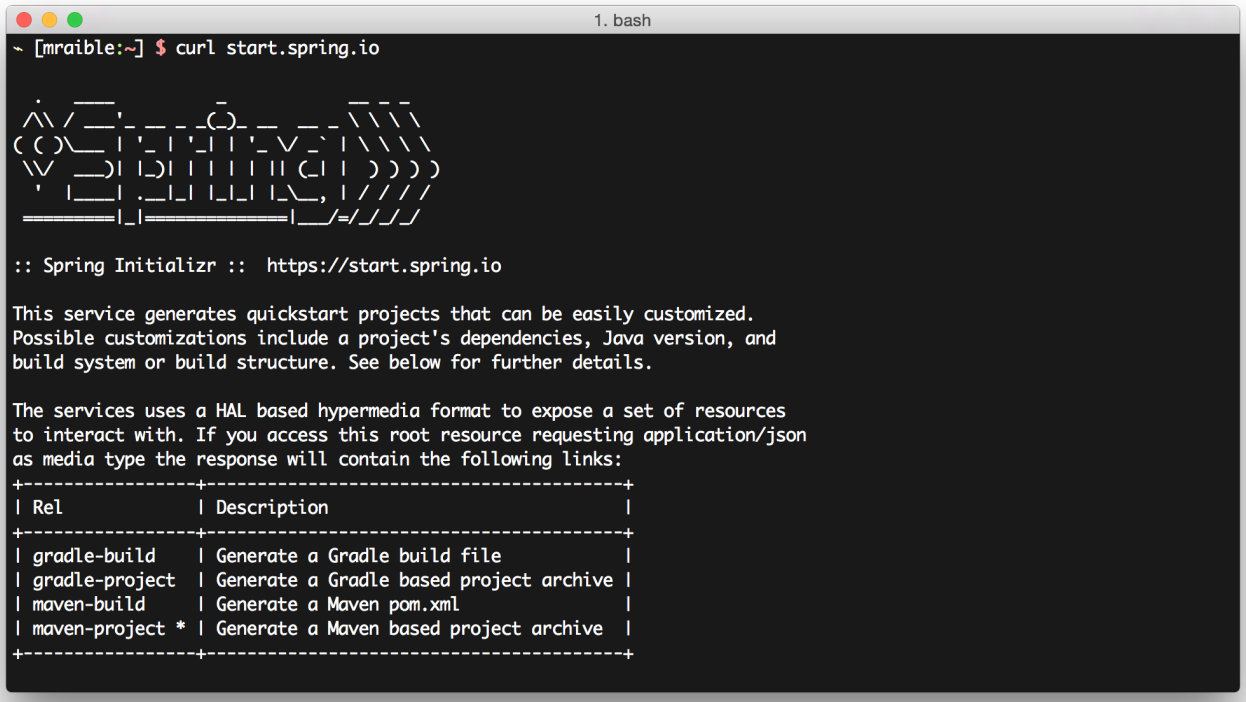


Figure 36. Spring Initializr via curl

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

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

Spring CLI

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

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

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

hello.groovy

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

To compile and run this application, simply type:

```
spring run hello.groovy
```

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

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

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

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

10 directories, 4 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.

```
2015-09-21 06:00:07.056  INFO 6140 --- [           main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
[Blog{id=1, name='First'}, Blog{id=2, name='Second'}]
2015-09-21 06:00:07.210  INFO 6140 --- [           main] demo.DemoApplication
: Started DemoApplication in 4.794 seconds (JVM running for 5.238)
```

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.
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.
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.

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 peaking 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.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.JolokiaAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.ManagementSecurityAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.ManagementServerPropertiesAutoConfiguratio
n,\
org.springframework.boot.actuate.autoconfigure.MetricFilterAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.MetricRepositoryAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.PublicMetricsAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.TraceRepositoryAutoConfiguration,\
org.springframework.boot.actuate.autoconfigure.TraceWebFilterAutoConfiguration
```

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:

- `/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`.
- `/health` - Returns information about application health.
- `/info` - Returns basic application info.
- `/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 classpaths. The boot-starter dependencies in the 21-Points Health application are as follows:

```
spring-boot-actuator
spring-boot-autoconfigure
spring-boot-loader-tools
spring-boot-starter-logging
spring-boot-starter-aopspring-boot-starter-data-jpa
spring-boot-starter-data-elasticsearch
spring-boot-starter-security
spring-boot-starter-web
spring-boot-starter-websocket
spring-boot-starter-thymeleaf
spring-cloud-cloudfoundry-connector
spring-cloud-spring-service-connector
spring-cloud-localconfig-connector
spring-security-config
spring-security-data
spring-security-web
spring-security-messaging
```

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 around 800 lines of XML. With Gradle, you end up with nine `*.gradle` files. However, their Groovy code adds up to only 344 lines.

Apache Maven calls itself 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, in which they provided a timeline of build tools from 1977 through 2013.

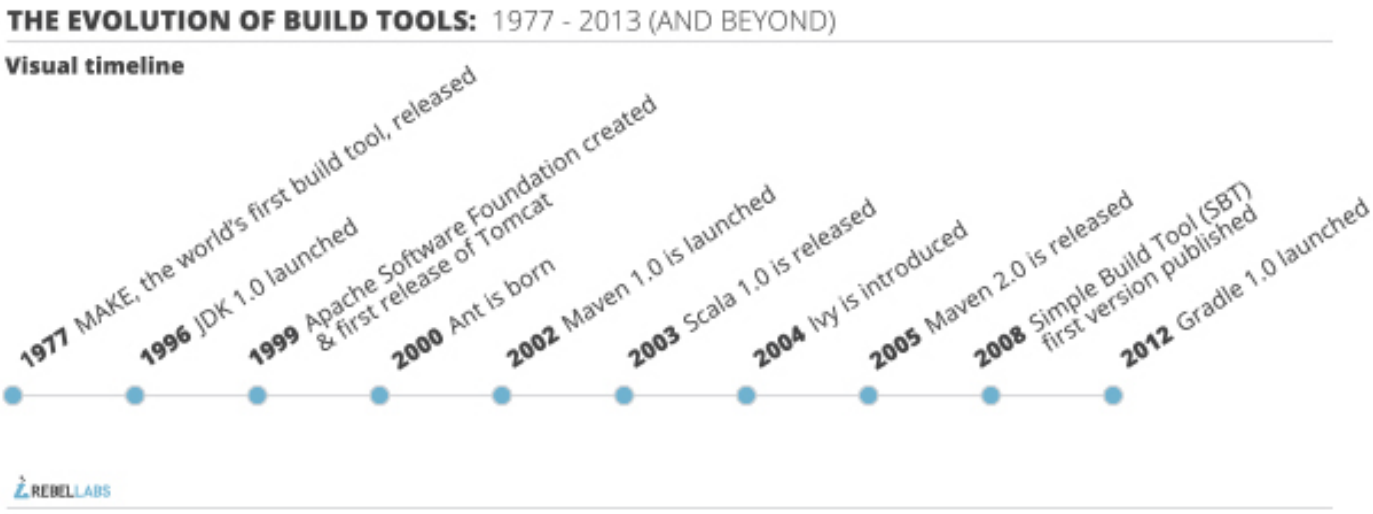


Figure 37. The Evolution of Build Tools, 1977-2013

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>0.1.10</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 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.

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.

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.

The JHipster documentation includes guides for configuring Eclipse and IntelliJ IDEA.

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:

- Requires authentication to every URL in your application.
- Generates a login form for you.
- Allows user:password to authenticate with form-based authentication.
- Allows the user to logout.
- Prevents CSRF attacks.
- Protects against session fixation.
- Security-header integration.
  - HTTP Strict Transport Security for secure requests.
  - X-Content-Type-Options integration.
  - Cache control.
  - X-XSS-Protection integration.
  - X-Frame-Options integration to help prevent clickjacking.
- Integrates with HttpServletRequest API methods: `getRemoteUser()`, `getUserPrinciple()`, `isUserInRole(role)`, `login(username, password)`, and `logout()`

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

- **HTTP Session Authentication** — Uses the HTTP session, so it is a stateful mechanism. Recommended for small applications.
- **OAuth2 Authentication** — A stateless security mechanism. You might prefer it if you want to scale your application across several machines.
- **Token-based authentication** — Like OAuth2, a stateless security mechanism. This is specific to JHipster, not provided by Spring Security.

It's possible that JHipster will soon support four authentication options. JHipster Team member, Thibaut Mottet, has been working on integrating Spring Social and has published a sample application with Google and Facebook authentication.

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** — An 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, 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. You can also use H2 for development, but then you'll lose out on Liquibase's "diff" feature.

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 Dynamo Database) 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 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 Mongeez 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 has more limitations and doesn't have a Liquibase equivalent. For example, it only works with Java 8 and it does not support OAuth2 authentication.

Liquibase

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

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

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

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

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

JHipster's development guide recommends the following workflow:

1. Modify your JPA entity (add a field, a relationship, etc.).
2. Run `mvn compile liquibase:diff`.
3. A new changelog is created in your `src/main/resources/config/liquibase/changelog` directory.

4. Review this changelog and add it to your `src/main/resources/config/liquibase/master.xml` file, so it is applied the next time you run your application.

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

Elasticsearch

Elasticsearch adds searchability to your entities. JHipster's Elasticsearch support requires choosing Java 8+ and 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 `grunt 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, OpenShift, 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. Recently, 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 10 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 January 2016).

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 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.