

# Full Stack Development with JHipster

**Second Edition**

Build full stack applications and microservices with Spring Boot  
and modern JavaScript frameworks



**Packt**

[www.packt.com](http://www.packt.com)

Deepu K Sasidharan and Sendil Kumar N

# **Full Stack Development with**

## **JHipster**

### ***Second Edition***

Build full stack applications and microservices with Spring Boot and modern JavaScript frameworks

**Deepu K Sasidharan**  
**Sendil Kumar N**

**Packt**

BIRMINGHAM - MUMBAI

# Full Stack Development with JHipster

## *Second Edition*

Copyright © 2020 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Commissioning Editor:** Kunal Chaudhari

**Acquisition Editor:** Alok Dhuri

**Content Development Editor:** Pathikrit Roy

**Senior Editor:** Afshaan Khan

**Technical Editor:** Gaurav Gala

**Copy Editor:** Safis Editing

**Project Coordinator:** Francy Puthiry

**Proofreader:** Safis Editing

**Indexer:** Tejal Daruwale Soni

**Production Designer:** Arvindkumar Gupta

First published: March 2018

Second edition: January 2020

Production reference: 1220120

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83882-498-3

[www.packt.com](http://www.packt.com)

*To my mother, Latha Kumari, and my father, K Sasidharan, for making me who I am.  
To my loving wife, Sabitha, for being supportive and patient throughout our journey together.  
To my family, friends, colleagues, and the JHipster community.*

*– Deepu K Sasidharan*

*To Nellaiyapen, Amutha, Sakthi, and Sahana for their advice, patience, and faith.  
To my amigos and all the awesome full stack developers out there.*

*– Sendil Kumar N*



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.packt.com](http://www.packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub.com](mailto:customercare@packtpub.com) for more details.

At [www.packt.com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

# Foreword

The first time I used JHipster was in 2013. I was an independent consultant at the time, and I used JHipster to demonstrate to my client how it could quickly generate an AngularJS and Spring Boot application. I liked the project so much that I decided to write a mini-book about JHipster for InfoQ. As part of the writing process, I developed a sample application and found bugs in JHipster. I reported these bugs to the project, sometimes with fixes.

JHipster is an incredible project. It gives Java developers the opportunity to generate applications that use modern JavaScript frameworks for their UI, while also using the Spring frameworks they know and love in the backend. The project started as an application generator and has gradually morphed into a development platform that makes it possible to create, build, and deploy both monoliths and microservices. Not only that, it follows many of its dependent project's best practices. Simply put, it creates code for you, allowing you to concentrate on your business logic, and gives you a fantastic development experience.

Deepu has been a joy to work with ever since I started reporting issues to JHipster. Not only does he know Angular and React exceptionally well, he also knows the internals of Yeoman and Node, which helps the project maintain momentum. Whenever there's a new major version of JHipster, Deepu is the one who seems to work the hardest and commit the most code.

I met Sendil Kumar N through the JHipster project as well. Sendil was an instrumental figure in migrating from AngularJS to Angular (initially called Angular 2) in 2016 and optimizing our webpack configuration. He also created JHipster's Kotlin support and continues to amaze me with his coding skills.

Deepu and Sendil are staples in the JHipster community, and this book is an in-depth guide to all things JHipster. They have gone the extra mile to deliver an exceptional book, and I think you'll enjoy learning from them.

I hope you have fun becoming a hip Java developer and building awesome apps with JHipster!

## Matt Raible

Web Developer, Java Champion, and Developer Advocate at Okta  
Denver, Colorado, USA

# Contributors

## About the authors

**Deepu K Sasidharan** is the co-lead of JHipster. He has been part of the core JHipster team since its inception and is an active contributor to the project. He currently works for XebiaLabs, a DevOps software company, as a senior polyglot product developer. Prior to that, he worked at TCS as a technical consultant focusing on innovative solutions for airlines. He has over 10 years of experience in the architecture, design, and implementation of enterprise web applications, and pre-sales. He is also a Java, JavaScript, web technology, and DevOps expert. When not coding, he likes to read about astronomy and science.

*First and foremost, I would like to thank my wife, Sabitha, for her patience and support. I would also like to thank Sendil Kumar, Julien Dubois, Antonio Goncalves, and the JHipster team for their support. Last but not least, I would like to thank the entire Packt editorial team for supporting me in this endeavor.*

**Sendil Kumar N** is part of the JHipster and Webpack team. He is an avid open source enthusiast and a contributor to many open source projects. He likes to explore and experiment with newer technologies and programming languages. He is passionate about (re)learning. He currently works at Uber as a senior software engineer, where he enhances the payment experience for Uber on the web. Before that, he designed, developed, and maintained enterprise products and DevOps tools that design and orchestrate releases for enterprises.

*Thanks to my wife, Sakthi, and daughter, Sahana, for their love and support. I would also like to thank Deepu K Sasidharan, Julien Dubois, Antonio Goncalves, and the entire JHipster team for their support and this awesome product. Finally, thanks to the Packt team, who were helpful and encouraging.*

## About the reviewer

**Julien Dubois** is the creator and lead developer of JHipster. He is also a Java Champion, with more than 20 years of experience in Java and web technologies.

Julien works as a Cloud Developer Advocate at Microsoft, focusing on improving Java and Spring support on Microsoft Azure.

*I would like to thank my wife, Aurélie, and my children, Gabrielle, Adrien, Alice, and Benjamin, for their patience while I was reviewing this book and developing JHipster.*

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](http://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

# Table of Contents

<b>Preface</b>	1
<hr/>	
<b>Section 1: Getting Started with the JHipster Platform</b>	
<b>Chapter 1: Introduction to Modern Web Application Development</b>	9
<b>Modern full stack web development</b>	10
<b>Web architecture patterns</b>	12
Monolithic web architecture	13
Microservice architecture	16
<b>Choosing the right pattern</b>	19
When to choose a monolithic architecture	19
When to choose a microservice architecture	20
<b>Summary</b>	21
<b>Chapter 2: Getting Started with JHipster</b>	22
<b>Why JHipster?</b>	23
<b>Goal and adoption of JHipster</b>	24
<b>Introduction to available technologies</b>	25
Client-side technologies	25
HTML5 and CSS3	26
SASS	26
Bootstrap	27
MVVM framework	27
Angular	28
React	28
Vue.js	29
Build tools – webpack and BrowserSync	29
Testing tools – Jest and Protractor	29
Internationalization	30
Server-side technologies	30
Spring Framework	31
Spring Boot	31
Spring Security	32
Spring MVC	32
Spring Data	32
Security	33
JWT	33
Session	33
OAuth2/OIDC	33
Build tools – Maven and Gradle	34
Hibernate	34
Liquibase	35
Caching	35

---

*Table of Contents*

Ehcache	35
Hazelcast	36
Infinispan	36
Memcached	36
Redis	36
Swagger	36
Thymeleaf	37
Micrometer	37
WebSocket	37
Kafka	37
Testing frameworks – JUnit, Gatling, and Cucumber	38
Introduction to database options	38
SQL databases	39
H2	39
MySQL	39
MariaDB	39
PostgreSQL	40
MS SQL	40
Oracle	40
NoSQL databases	40
MongoDB	40
Cassandra	41
Couchbase	41
Elasticsearch	41
<b>Installing and setting up JHipster</b>	41
Prerequisites	42
Tools required	42
Java	42
Git	43
Node.js	43
Docker	44
IDE configuration	45
System setup	45
Installation of JHipster	46
<b>Summary</b>	47
<b>Section 2: Building and Customizing Web Applications with JHipster</b>	
<b>Chapter 3: Building Monolithic Web Applications with JHipster</b>	49
<b>Application generation</b>	50
Step 1 – preparing the workspace	50
Step 2 – generating code using JHipster	50
Server-side options	52
Client-side options	56
Internationalization options	58
Testing	59
Modules	59
<b>Code walkthrough</b>	61
File structure	62

---

Server-side source code	64
Java source	66
Resources	73
Client-side source code	74
<b>Starting the application</b>	77
<b>Application modules</b>	78
Home and login modules	78
Account modules	81
Settings	81
Password	82
Registration	82
Admin module	83
User management	84
Metrics	84
Health	85
Configuration	86
Audits	86
Logs	87
API	88
<b>Running generated tests</b>	88
Server-side tests	89
Client-side tests	89
<b>Summary</b>	90
<b>Chapter 4: Entity Modeling with JHipster Domain Language</b>	91
<b>Introduction to JDL</b>	91
DSL grammar for JDL	92
Entity modeling with JDL	93
Relationship management	94
DTO, service, and pagination options	95
JDL-Studio	96
Use case entity model	98
Entities	98
Relationships	101
Options for entities	102
<b>Entity generation with JHipster</b>	104
Generated code walkthrough	105
Server-side source code	107
Domain class for the entity	107
Repository interface for the entity	110
Service class for the entity	110
Resource class for the entity	111
Client-side	112
TypeScript model class for the entity	112
Angular services for the entity	113
Angular components of the entity	113
Angular route for the entity	114
Angular module for the entity	115
<b>Generated pages</b>	115

<b>Running the generated tests</b>	118
<b>Summary</b>	118
<b>Chapter 5: Customization and Further Development</b>	119
<b>Live reload for development</b>	120
Spring Boot DevTools	120
Webpack dev server and BrowserSync	121
Setting up live reloads for an application	122
<b>Customizing the Angular frontend for an entity</b>	122
Bringing back the sorting functionality	129
Adding a filtering functionality	131
<b>Editing an entity using the JHipster entity sub-generator</b>	133
<b>Changing the look and feel of the application</b>	141
<b>Adding a new i18n language</b>	143
<b>Authorization with Spring Security</b>	145
Limiting access to entities	145
Limiting access to create/edit/delete entities	146
Limiting access to the data of other users	147
<b>Summary</b>	150
<b>Section 3: Continuous Integration and Testing</b>	
<b>Chapter 6: Testing and Continuous Integration</b>	152
<b>Fixing and running tests</b>	153
<b>Briefing on CI</b>	158
<b>CI/CD tools</b>	159
Jenkins	160
Azure Pipelines	160
Travis CI	160
GitLab CI	161
GitHub Actions	161
<b>Setting up Jenkins</b>	161
<b>Creating a Jenkins pipeline using JHipster</b>	162
The Jenkinsfile and its stages	164
Setting up the Jenkinsfile in a Jenkins server	167
<b>Summary</b>	172
<b>Chapter 7: Going into Production</b>	173
<b>Introduction to Docker</b>	173
Docker containers	174
Dockerfiles	174
Docker Hub	175
Docker Compose	176
<b>Starting the production database with Docker</b>	177
<b>Introducing Spring profiles</b>	178

<b>Packaging the application for local deployment</b>	180
Building and deploying using Docker	180
Building and deploying an executable archive	182
<b>Upgrading to the newest version of JHipster</b>	182
<b>Deployment options supported by JHipster</b>	186
Heroku	186
Cloud Foundry	187
Amazon Web Services	187
Google App Engine	187
Azure Spring Cloud	187
Azure App Service	188
<b>Production deployment to the Heroku Cloud</b>	189
Summary	192

## **Section 4: Converting Monoliths to Microservice Architecture**

---

<b>Chapter 8: Microservice Server-Side Technologies</b>	194
<b>Microservice applications versus monoliths</b>	195
Scalability	195
Efficiency	196
Time constraint	196
<b>Building blocks of a microservices architecture</b>	197
Service registry	198
Service discovery	198
Client-side discovery pattern	199
Server-side discovery pattern	199
Health check	200
Dynamic routing and resiliency	200
Security	201
Fault tolerance and failover	201
<b>Options supported by JHipster</b>	202
JHipster Registry	202
Netflix Eureka	203
Spring Cloud Config Server	204
HashiCorp Consul	205
Service discovery	205
Health discovery	205
Key/Value store	206
Multiple data centers	206
JHipster gateway	206
Netflix Zuul	206
Hystrix	207
JHipster Console	208
Elasticsearch	208
Logstash	208

Kibana	209
Zipkin	209
Prometheus	209
JWT authentication	211
How JWT works	212
JHipster UAA server	212
<b>Summary</b>	214
<b>Chapter 9: Building Microservices with JHipster</b>	215
<b>Application architecture</b>	216
<b>Generating a microservice stack using JDL</b>	217
Application modeling using JDL	217
Gateway application	219
JDL specification for the gateway application	220
Microservice invoice application	222
JDL specification for the invoice application	222
Microservice notification application	222
JDL specification for the notification application	223
Modeling microservice entities in JDL	223
Application generation with import-jdl	228
<b>Gateway application</b>	230
Gateway application entities	234
<b>Invoice microservice configuration</b>	234
<b>Notification microservice configuration</b>	238
<b>Summary</b>	239
<b>Chapter 10: Working with Microservices</b>	240
<b>Setting up JHipster Registry locally</b>	240
Using a pre-packaged JAR file	241
Docker mode	244
<b>Running a generated application locally</b>	245
Gateway application pages	245
JHipster Registry pages	248
Dashboard	249
System status	249
Instances registered	250
General info and health	250
Application listing page	250
Metrics page	251
Health page	253
Configuration page	253
Logs page	254
Loggers	255
Swagger API endpoints	255
Running invoice and notification applications locally	260
Explaining the generated entity pages	262
<b>Summary</b>	264

## **Section 5: Deployment of Microservices**

---

<b>Chapter 11: Deploying with Docker Compose</b>	266
<b>Introducing microservice deployment options</b>	266
A short introduction to Docker Compose	267
Introduction to Kubernetes	269
Introduction to OpenShift	272
<b>Generated Docker Compose files</b>	272
Walking through the generated files	273
Building and deploying everything to Docker locally	277
<b>Generating Docker Compose files for microservices</b>	279
Features of the deployed application	283
JHipster Console demo	284
Scaling up with Docker	288
<b>Summary</b>	289
<b>Chapter 12: Deploying to the Cloud with Kubernetes</b>	290
<b>Generating Kubernetes configuration files with JHipster</b>	291
Generating the Kubernetes manifests	291
<b>Walking through the generated files</b>	297
<b>Deploying the application to Google Cloud with Kubernetes</b>	301
<b>Using Istio service mesh</b>	309
What is Istio?	309
Microservice with Istio service mesh	311
Deploying Istio to a Kubernetes cluster	314
Generating the application	316
Deploying to Google Cloud	318
<b>Summary</b>	323

## **Section 6: React and Vue.js for the Client Side**

---

<b>Chapter 13: Using React for the Client-Side</b>	325
<b>Generating an application with React client-side</b>	325
<b>Technical stack and source code</b>	328
Technical stacks	329
Using TypeScript	329
State management with Redux and friends	330
Routing with React Router	331
HTTP requests using Axios	333
Bootstrap components using reactstrap	333
Unit testing setup	334
Generated source code	334
<b>Generating an entity with React client-side</b>	340
<b>Summary</b>	347
<b>Chapter 14: Using Vue.js for the Client-Side</b>	348

<b>Generating an application with Vue.js client-side</b>	348
<b>Technical stack and source code</b>	351
Technical stack	352
Using TypeScript	352
State management with Vuex	352
Routing with Vue Router	353
HTTP requests using Axios	355
Bootstrap components using BootstrapVue	355
Unit testing setup	356
Generated source code	357
<b>Generating an entity with VueJS client-side</b>	362
<b>Summary</b>	369
<b>Chapter 15: Best Practices with JHipster</b>	370
<b>The next steps to take</b>	371
Adding a shopping cart for the application	371
Improving end-to-end tests	372
Improving the CI/CD pipeline	373
Create an e-commerce application with React or Vue.js	373
Building a JHipster module	374
<b>The best practices to keep in mind</b>	374
Choosing a client-side framework	375
Choosing a database option	375
Architecture considerations	377
Security considerations	378
Deployment and maintenance	378
General best practices	380
<b>Using JHipster modules</b>	380
<b>Contributing to JHipster</b>	382
<b>Summary</b>	382
<b>Other Books You May Enjoy</b>	383
<b>Index</b>	386

# Preface

This book, *Full Stack Development with JHipster*, aims to address the following challenges faced by full stack developers today:

- There is a multitude of technologies and options out there to learn.
- Customer demands have increased, and hence time to market has become more stringent.
- Client-side frameworks have become complicated and difficult to integrate.
- There is so much integration between technologies and concepts that it overwhelms most novice and even proficient developers.

JHipster provides a platform for developers to easily create web applications and microservices from scratch, without having to spend a lot of time wiring everything together and integrating technologies. This frees up time immensely for developers to actually focus on their solution rather than spending time learning and writing boilerplate code. JHipster will help novice and experienced developers to be more productive from day one. It's like pair programming with an entire community.

This book will take you on a journey from zero to hero in full stack development. You will learn to create complex production-ready Spring Boot and Angular web applications from scratch using JHipster, and you will go on to develop features and business logic and deploy it on cloud services. You will also learn about microservices and how to convert a monolithic application in the microservice architecture as it evolves using JHipster. Finally, you will deploy microservices to a cloud provider using Kubernetes. Additionally, you will learn how to make use of the React and Vue.js support in JHipster and about various best practices and suggestions from the JHipster community and the core development team.

## Who this book is for

Anyone with a basic understanding of building Java web applications and basic exposure to Spring and Angular/React/Vue.js will benefit from using this book to learn how to use JHipster for cutting-edge full stack development or to improve their productivity by cutting down boilerplate and learning new techniques. The audience can be broadly classified as follows:

- Full stack web app developers who want to reduce the amount of boilerplate they write and save time, especially for greenfield projects

- Backend developers who want to learn full stack development with Angular, React, or Vue.js
- Full stack developers who want to learn microservice development
- Developers who want to jump-start their full stack web application or microservice development
- Developers who want to quickly prototype web applications or microservices
- Developers who want to learn about Java microservices and their deployment with Kubernetes

## What this book covers

Chapter 1, *Introduction to Modern Web Application Development*, introduces two widely used full stack web application development architectures. It also lays out commonly faced challenges in full stack web application development.

Chapter 2, *Getting Started with JHipster*, introduces the JHipster platform. It will also give the reader a brief overview of different server-side, client-side, and database technology options offered by JHipster. This chapter will also provide instructions to install and use JHipster, and the various tools and options it supports.

Chapter 3, *Building Monolithic Web Applications with JHipster*, guides the user through the creation of production-ready Spring Boot and Angular web applications from scratch using JHipster and will take the reader through the generated code, screens, and concepts.

Chapter 4, *Entity Modeling with JHipster Domain Language*, introduces the reader to **JHipster Domain Language (JDL)** and will teach you about building business logic with entity modeling and entity creation using JDL and JDL-Studio.

Chapter 5, *Customization and Further Development*, guides the reader through further development of the generated application. It will also teach the reader more about using technologies such as Angular, Bootstrap, Spring Security, Spring MVC REST, and Spring Data.

Chapter 6, *Testing and Continuous Integration*, guides the reader through testing and setting up a continuous integration pipeline using Jenkins.

Chapter 7, *Going into Production*, shows the reader how to use Docker and how to build and package the app for production. It will also introduce the reader to some of the production cloud deployment options supported by JHipster.

Chapter 8, *Microservice Server-Side Technologies*, gives an overview of different options available in the JHipster microservice stack.

Chapter 9, *Building Microservices with JHipster*, guides the reader through converting a JHipster monolith web application into a full-fledged microservice architecture with a gateway, registry, monitoring console, and multiple microservices. It will also guide the reader through creating domain entities for the microservice architecture using JDL. It will also guide the reader through the generated code.

Chapter 10, *Working with Microservices*, guides the reader through running the generated applications locally. It will also guide the reader through the generated components, such as the JHipster Registry, JHipster Console, and API gateways.

Chapter 11, *Deploying with Docker Compose*, introduces the reader to advanced local and cloud deployment options for microservices. It will also guide the user through local deployment and testing of the generated microservice stack using Docker Compose and JHipster.

Chapter 12, *Deploying to the Cloud with Kubernetes*, guides the user through the Google Cloud deployment of the generated microservice stack using Kubernetes and JHipster. It will also introduce Istio and guide the reader through creating and deploying microservices with Istio on Kubernetes.

Chapter 13, *Using React for the Client-Side*, takes the user through generating an application with React on the client-side instead of Angular using JHipster.

Chapter 14, *Using Vue.js for the Client-Side*, takes the user through generating an application with Vue.js on the client-side instead of Angular using JHipster.

Chapter 15, *Best Practices with JHipster*, summarizes what the reader has learned so far and will suggest best practices and next steps to utilize the skills learned.

# To get the most out of this book

To get the most out of this book, you will need to know the basics of the following technologies:

- Web technologies (HTML, JavaScript, and CSS)
- Java 8
- The Spring Framework
- SQL databases
- Build tools (Maven or Gradle)
- npm

It will also be helpful if you are familiar with using technologies such as Docker and Kubernetes, as it will help you make sense of some of the chapters.

You will also need JDK 11, Git, Docker, and Node.js installed; your favorite web browser; a terminal application; and your favorite code editor/IDE.

## Download the example code files

You can download the example code files for this book from your account at [www.packtpub.com](https://www.packtpub.com). If you purchased this book elsewhere, you can visit <https://www.packtpub.com/support> and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at [www.packtpub.com](https://www.packtpub.com).
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Full-Stack-Development-with-JHipster-Second-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: [https://static.packt-cdn.com/downloads/9781838824983\\_ColorImages.pdf](https://static.packt-cdn.com/downloads/9781838824983_ColorImages.pdf).

## Conventions used

There are a number of text conventions used throughout this book.

**CodeInText:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "In the backend, modify the `save` method of `ProductOrderService.java` to create an invoice and shipment for the `ProductOrder` and save them all."

A block of code is set as follows:

```
entity Product {  
    name String required  
    description String  
    price BigDecimal required min(0)  
    size Size required  
    image ImageBlob  
}  
  
enum Size {  
    S, M, L, XL, XXL  
}  
  
entity ProductCategory {  
    name String required  
    description String  
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
entity ProductOrder {  
    placedDate Instant required  
    status OrderStatus required  
    invoiceId Long  
    code String required  
}
```

Any command-line input or output is written as follows:

```
> cd invoice  
> ./gradlew
```

**Bold:** Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "You can alternatively test this via your **Gateway** application. Log in to our **Gateway** application and then navigate to **Administration | Gateway**."

Warnings or important notes appear like this.



Tips and tricks appear like this.



## Get in touch

Feedback from our readers is always welcome.

**General feedback:** Email [feedback@packtpub.com](mailto:feedback@packtpub.com) and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at [questions@packtpub.com](mailto:questions@packtpub.com).

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit <https://www.packtpub.com/support/errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details.

**Piracy:** If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com).

## Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit [packtpub.com](http://packtpub.com).

# 1

## Section 1: Getting Started with the JHipster Platform

In this section, you will be introduced to the modern web application development architecture and the JHipster platform. Here, you will learn about the two widely used full stack web application development architectures and have a brief overview of different server-side, client-side, and database technology options.

This section comprises the following chapters:

- Chapter 1, *Introduction to Modern Web Application Development*
- Chapter 2, *Getting Started with JHipster*

# 1

# Introduction to Modern Web Application Development

According to the Stack Overflow developer survey 2019 (<https://insights.stackoverflow.com/survey/2019#developer-roles>), **full stack developer** is the most popular developer title. The software industry defines a full stack developer as someone who can work on different areas of an application stack. The term *stack* refers to the different components and tools that make up an application.

In terms of web application development, the stack can be broadly classified into two areas—**frontend** and **backend** stack, also known as the **client-side** and **server-side** stack. The term *frontend* generally refers to the part of the code that is responsible for the user interface, and the term *backend* refers to the part that is responsible for the business logic, database interactions, user authentication, server configuration, and so on. There is also the **DevOps** part of the application, which includes continuous integration, production deployment, and so on. A full stack Java web application developer is expected to work on both frontend and backend technologies, ranging from writing HTML/JavaScript for the user interface to writing Java class files for business logic and SQL queries for database operations. They are also expected to work on DevOps, ranging from production deployments to setting up **continuous integration and continuous delivery (CI/CD)** as required.

With an ever-evolving software architecture landscape, the scope of technologies that a full stack web developer is expected to work with has increased dramatically. It is no longer enough that we can write HTML and JavaScript to build a user interface—we are expected to know client-side frameworks, such as Angular, React, and Vue.js. It is also not enough that we are proficient in enterprise Java and SQL—we are expected to know server-side frameworks, such as Spring, Hibernate, Play, and Quarkus.

In this chapter, we will introduce the following topics:

- Modern full stack web development
- Web architecture patterns
- Choosing the right pattern

## Modern full stack web development

The life of a full stack developer would be worthy of a whole book by itself, so let's leave that topic for another day.

Instead, let's look at a user story from a full stack Java web application and see what is involved.

Let's use an example of developing a user management module for a typical Java web application. Let's assume that you would be writing unit test cases for all of the code, and so we won't look at it in detail here:

1. You would start by designing the architecture for the feature. You would decide on the plugins and frameworks to use, patterns to follow, and so on.
2. You will be modeling the domain model for the feature depending on the database technology used.
3. Then, you would create server-side code and database queries to persist and fetch data from the database.
4. Once the data is ready, you would implement server-side code for any business logic.
5. Then, you would implement an API that can be used to provide data for the presentation over an HTTP connection.
6. You would write integration tests for the API.
7. Since the backend is ready, you would start writing frontend code in JavaScript or similar technology.
8. You would write client-side services to fetch data from the backend API.
9. You would write client-side components to display the data on a web page.
10. You would build the page and style it as per the design provided.
11. You would write some automated end-to-end tests for the web page.
12. You are not done yet. Once you have tested whether everything works locally, you would create pull requests or check the code into the version control system used.

13. You would wait for the continuous integration process to verify everything and fix anything that is broken.
14. Once everything is green and the code is accepted, you would typically start the deployment of this feature to a staging or acceptance environment, either on-premises or to a cloud provider using technologies like Docker and Kubernetes. If you choose the latter, you would be expected to be familiar with the cloud technologies used as well. You would also be upgrading the database schema as necessary and writing migration scripts when required.
15. Once the feature is accepted, you might be responsible for deploying it into the production environment in a similar way, troubleshooting issues where necessary. In some teams, you might swap the steps with other team members so that you would be deploying a feature developed by your coworker while they deploy yours.
16. You might also be responsible, along with your coworkers, for making sure that the production environment is up and running, including the database, virtual machines, and so on.

As you can see, it is no easy task. The range of responsibilities spans from making stylesheet updates on the client-side to running database migration scripts on a virtual machine in the production cloud service. If you are not familiar enough with the setup, then this would be a herculean task, and you would soon be lost in the vast ocean of frameworks, technologies, and design patterns out there.

Full stack development is not for the faint-hearted. It takes a lot of time and effort to keep yourself up to date with the various technologies and patterns in multiple disciplines of software development. The following are some of the common problems you might face as a full stack Java developer:

- Client-side development is not just about writing plain HTML and JavaScript anymore. It is becoming as complex as server-side development, with build tools, transpilers, frameworks, and patterns.
- There is a new framework almost every week in the JavaScript world, and if you are coming from a Java background, it could be very overwhelming for you.
- Container technologies such as Docker revolutionized the software industry, but they also introduced a lot of new stuff to learn and keep track of, such as orchestration tools and container management tools.
- Cloud services are growing day by day. To stay on track, you would have to familiarize yourself with their APIs and related orchestration tools.

- Java server-side technologies have also undergone a major shift in recent times with the introduction of JVM languages, such as Scala, Groovy, and Kotlin, forcing you to keep yourself up to date with them. On the other side, server-side frameworks are becoming more feature-rich, and therefore more complex.

The most important thing of all is to make sure that all of these work well together when required. This task will need a lot of configuration, some glue code, and endless cups of coffee.



**Transpilers** are source-to-source compilers. Whereas a traditional compiler compiles from source to binary, a transpiler compiles from one type of source code to another type of source code. TypeScript and CoffeeScript are excellent examples of this; both compile down to JavaScript.

It's very easy to get lost here, and this is where technologies such as JHipster and Spring Boot step in to help. We will look at the details of these technologies in later chapters, but in short, they help by providing the wiring between moving parts so that you only need to concentrate on writing business code. JHipster also helps by providing the abstractions to deploy and manage the application to various cloud providers.

## Web architecture patterns

The full stack landscape is further complicated by the different web architecture patterns commonly used these days. The widely used web application architecture patterns today can be broadly classified into two patterns—**monolithic architecture** and **microservice architecture**, the latter of which has become mainstream (fashionable) in recent years.

## Monolithic web architecture

A monolithic architecture is the most widely used pattern for web applications because of its simplicity to develop and deploy. Though the actual moving parts will differ from application to application, the general pattern remains the same. In general, a monolithic web application can do the following:

- It can support different clients, such as desktop/mobile browsers and native desktop/mobile applications.
- It can expose APIs for third-party consumption.
- It can integrate with other applications over REST/SOAP web services or message queues.
- It can handle HTTP requests, execute business logic, access databases, and exchange data with other systems.
- It can run on web application containers, such as Tomcat and JBoss.
- It can be scaled vertically by increasing the power of the machines it runs on or scaled horizontally by adding additional instances behind load balancers.

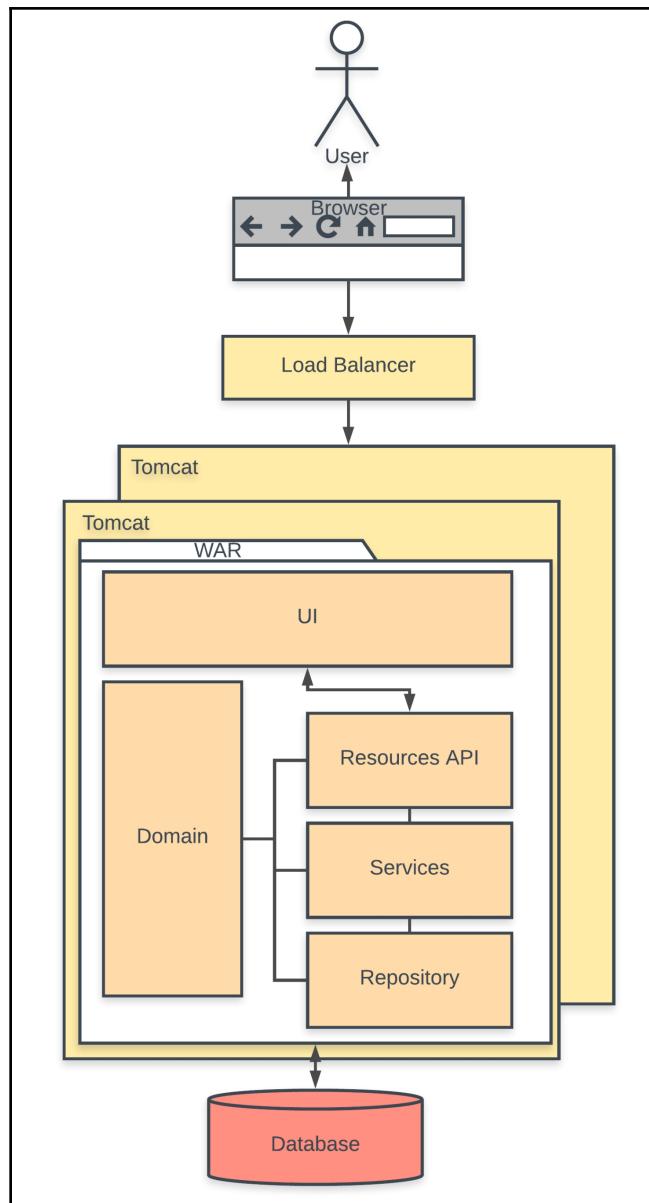


**REST** (short for **REpresentational State Transfer**) relies on a stateless, client-server, cacheable communications protocol. HTTP is the most commonly used protocol for REST. It is a lightweight architectural style in which RESTful HTTP communication is used to transfer data between a client and a server, or between two systems.

**SOAP** (short for **Simple Object Access Protocol**) is a messaging protocol using HTTP and XML. It is widely used in SOAP web services to transfer data between two different systems.

An example of a typical monolithic web application architecture would be as follows: Let's imagine an online hotel reservation system that takes online reservation orders from customers, verifies the room availability, verifies the payment option, makes the reservation, and notifies the hotel. The application consists of several layers and components, including a client-side app—which builds a nice, rich user interface—and several other backend components responsible for managing the reservations, verifying the payment, notifying customers/hotels, and so on.

The application will be deployed as a single monolithic **web application archive (WAR)** file that runs on a web application container such as Tomcat, and will be scaled horizontally by adding multiple instances behind an Apache web server acting as a load balancer. Take a look at the following diagram:



The advantages of a monolithic web application architecture are as follows:

- It is simpler to develop, as the technology stack is uniform throughout all layers.
- It is simpler to test, as the entire application is bundled in a single package, making it easier to run integration and end-to-end tests.
- It is simpler and faster to deploy, as you only have one package to worry about.
- It is simpler to scale, as you can multiply the number of instances behind a load balancer to scale it out.
- It requires a smaller team to maintain the application.
- The team members share more or less the same skill set.
- The technical stack is simpler and, most of the time, is easier to learn.
- Initial development is faster, thereby making the time to market shorter.
- It requires a simpler infrastructure. Even a simple application container or JVM will be sufficient to run the application.

The disadvantages of a monolithic web application architecture are as follows:

- Components are tightly coupled together, resulting in unwanted side effects, such as changes to one component causing a regression in another.
- It becomes complex and huge as time passes, resulting in slow development turnaround. New features will take more time to develop and refactoring existing features will be more difficult because of tight coupling.
- The entire application needs to be redeployed whenever any change is made.
- It is less reliable because of tightly coupled modules. A small issue with any service might break the entire application.
- Newer technology adoption is difficult as the entire application needs to be migrated. Incremental migration is not possible most of the time, which means that many monolithic applications end up having an outdated technology stack after some years.
- Critical services cannot be scaled individually, resulting in increased resource usage. As a result, the entire application will need to be scaled.
- Huge monolith applications will have a higher start up time and higher resource usage in terms of CPU and memory.
- Teams will be more interdependent and it will be challenging to scale the teams.

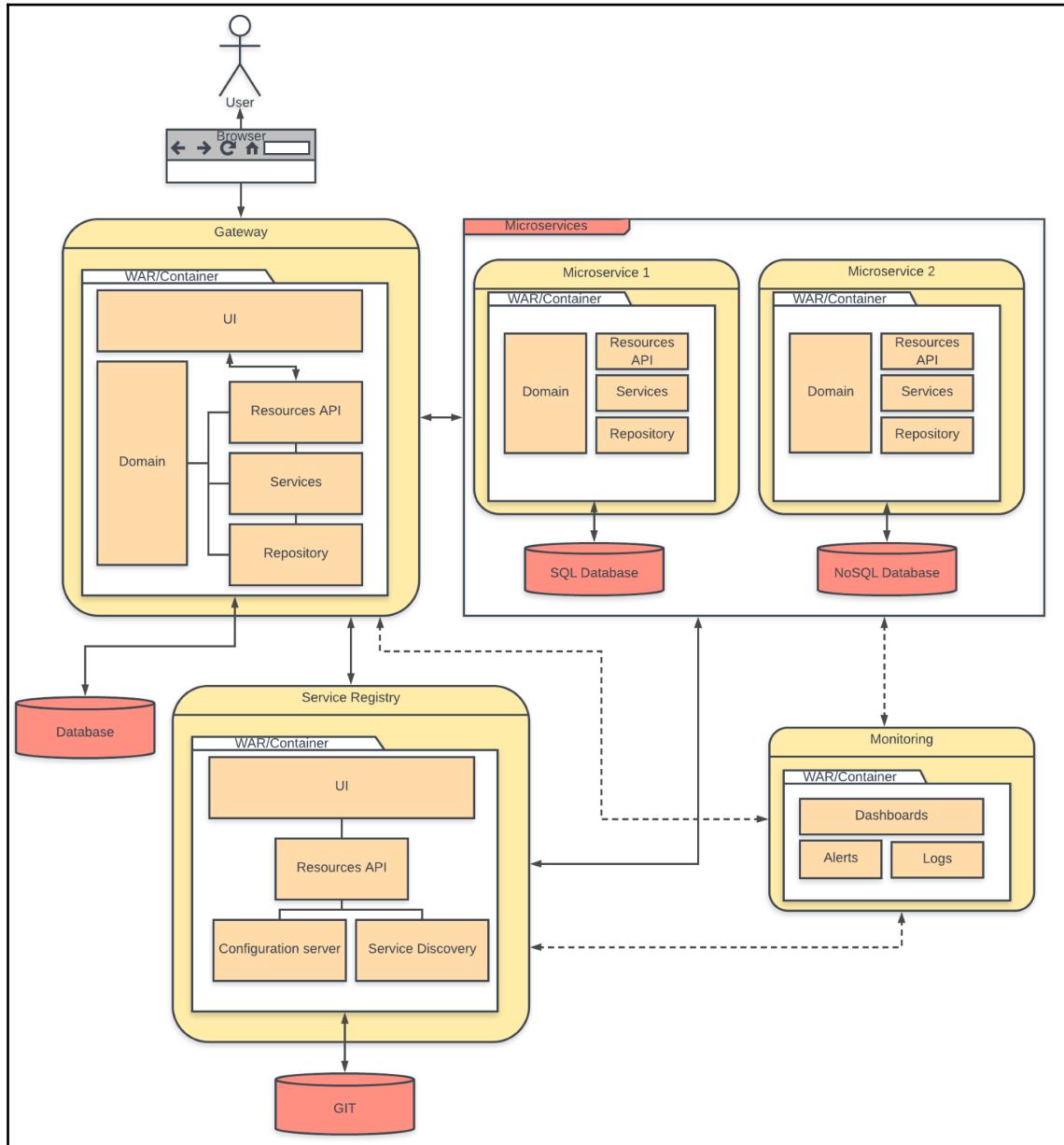
## Microservice architecture

The microservice architecture has gained momentum in recent years and is gaining popularity in large-scale web application development because of its modularity and scalability. Microservice architecture can offer almost all the features of a monolith that we saw in the previous section. Additionally, it offers many more features and flexibility and so is often considered a superior choice for complex, large-scale applications. Unlike the monolithic architecture, it's quite difficult to generalize the microservice architecture as it could vary heavily depending on the use case and implementation. But they do (generally) share some common traits, which are as follows:

- Microservice components are loosely coupled. Components can be developed, tested, deployed, and scaled independently without disrupting other components.
- Components need not be developed using the same technology stack. This means that a single component can choose its own technology stack and programming language.
- They often utilize advanced features such as service discovery, circuit breaking, and load balancing.
- Microservice components are mostly lightweight and perform a specific functionality. For example, an authentication service will only care about authenticating a user into the system.
- It often has an extensive monitoring and troubleshooting setup.

An example of a microservice web application architecture would be as follows: Let's imagine a huge online e-commerce system where customers can go through categories of merchandise, list their favorites, add items to a shopping cart, make and track orders, and so on. The system has inventory management, customer management, multiple payment modes, order management, and so on. The application consists of several modules and components, including a UI gateway application, which builds a nice, rich user interface and also handles user authentication and load balancing, and several other backend applications responsible for managing the inventory, verifying the payment, and managing orders. It also has performance monitoring and automatic failover for services.

The application will be deployed as multiple executable WAR files in Docker containers hosted by a cloud provider. Take a look at the following diagram:



The advantages of a microservice web application architecture are as follows:

- Loosely coupled components resulting in better isolation, which means that they are easier to test and faster to start up.
- Faster development turnaround and better time to market for new features, and existing features can be easily refactored.
- Services can be deployed independently, making the application more reliable and making patching easier.
- Issues, such as a memory leak in one of the services, are isolated and will not bring down the entire application.
- Technology adoption is easier, as components can be independently upgraded with incremental migration, making it possible to have a different stack for each component.
- More complex and efficient scaling models can be established. Critical services can be scaled more effectively. Infrastructure is used more efficiently.
- Individual components will start up faster, making it possible to parallelize and improve overall startup for large systems.
- Teams will be less dependent on each other. This is best suited for agile teams.

The disadvantages of a microservice web application architecture are as follows:

- It is more complex in terms of the overall stack as different components might have different technology stacks, forcing the team to invest more time in keeping up with them.
- It is difficult to perform end-to-end tests and integration tests as there are more moving parts in the stack.
- The entire application is more complex to deploy as there are complexities with containers, orchestration, and virtualization involved.
- Scaling is more efficient, but it is also more complex, as it would require advanced features, such as service discovery and DNS routing.
- It requires a larger team to maintain the application, as there are more components and more technologies involved.
- Team members share varying skill sets based on the component they work on, making replacements and knowledge sharing harder.
- Applications with many microservices and different teams managing different services creates more organizational challenges.
- The technical stack is complex and, most of the time, it is harder to learn.
- Initial development time will be higher, making the time to market longer.

- It requires a complex infrastructure. Most often, it will require containers (Docker), orchestration (Kubernetes), and multiple JVM or app containers to run on.
- To effectively manage distributed systems, you need to set up monitoring, distributed tracing, service discovery, and so on. These are not easy to set up and require additional components to be run which will produce overhead and additional costs.

## Choosing the right pattern

When starting a new project these days, it is always difficult to choose an architecture pattern. There are so many factors to take into account, and it is easy to get confused with all the hype surrounding different patterns and technologies (see the blog post on **Hype Driven Development (HDD)**) at <https://blog.daftcode.pl/hype-driven-development-3469fc2e9b22>). The following are some general guidelines on when to choose a monolithic web application architecture over a microservice architecture and vice versa.

## When to choose a monolithic architecture

The following list can be used as a general guide when choosing a monolithic architecture. This is not a definitive list, but it gives you an idea of when to go with a monolithic architecture over a microservice architecture:

- When the **application scope** is small and well defined, and you are sure that the application will not grow tremendously in terms of features—for example, a blog, a simple online shopping website, or a simple CRUD application
- When the **team size** is small, say fewer than eight people (this isn't a hard limit, but one that's based on the practicality of the team size)
- When the **average skill set** of the team is either novice or intermediate
- When the initial **time to market** is critical
- When you do not want to spend too much on **infrastructure**, monitoring, and so on
- When your **user base** is rather small and you do not expect it to grow—for example, an enterprise app targeting a specific set of users

In most practical use cases, a monolithic architecture would suffice. Read the next section to see when you should consider a microservice architecture over a monolithic architecture.

## When to choose a microservice architecture

The following list can be used as a general guide to choosing a microservice architecture. This is not a definitive list, but gives you an idea of when to go with a microservice architecture over a monolithic architecture. Note that, unlike choosing a monolithic architecture, the decision here is more complex and may involve consideration of many of the following points:

- When the **application scope** is large and well defined and you are sure that the application will grow tremendously in terms of features—for example, an online e-commerce store, a social media service, a video streaming service with a large user base (yes, I'm thinking of Netflix), or an API provider
- When the **team size** is large and there are enough members to effectively develop individual components independently
- When the **average skill set** of the team is good and team members are confident about advanced microservice patterns
- When the initial **time to market** is not critical as the microservice architecture will take more time to get going initially
- When you are ready to spend more on **infrastructure, monitoring**, and so on, in order to improve the product quality
- When your **user base** is huge and you expect it to grow—for example, a social media application targeting users all over the world

Though a monolithic architecture would suffice in most cases, and you should choose it for simpler use cases, investing upfront in a microservice architecture will reap long-term benefits when the application scope is large and you know it will grow further. There is no silver bullet and there are always trade-offs.



For more on these architecture patterns, you can refer to a nice blog post at <https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59>.

# Summary

So far, we've learned what full stack development is and compared two of the most prominent architecture patterns. We also learned about the advantages and disadvantages of monolithic and microservice architecture, which helps us to choose the right pattern for our use case.

In the next chapter, we will take a deep dive into the JHipster platform and look at all the options it provides. We will also learn how to install JHipster and set up our tools and development environment.

# 2

# Getting Started with JHipster

JHipster is a development platform that helps you go from zero to hero! JHipster can help you to create beautiful web applications and complex microservices architectures in a jiffy. JHipster also offers various tools to develop the applications further using business entities, and deploy them to various cloud services and platforms. At its core, JHipster is a Yeoman code generator that creates Spring Boot and Angular/React/Vue.js-based applications. It can create monolithic architectures as well as microservices architectures with every feature working out of the box.

In this chapter, we will cover the following topics:

- Why you should use JHipster and how it helps compared to traditional development approaches
- The goal of JHipster
- The various server-side and client-side technology options available in JHipster
- Preparation of a development environment
- Installation of JHipster and required dependencies



Yeoman (<http://yeoman.io>) is a scaffolding tool that helps you to create code generators. You can use it to create any kind of application generator with the help of the built-in template engine and tools.

# Why JHipster?

If you are wondering why you should be using JHipster, then just imagine the following scenario. You are tasked to build a web application, let's say a blog with an Angular frontend and a Java backend, with features for users to create blog posts and be able to display blog posts based on user permissions. You are also asked to build administrative modules such as user management and monitoring. Finally, you have to test and deploy the application to a cloud service using Docker.

If you are approaching this challenge the traditional way, you will most probably be performing the following steps. Let's skip the details for simplicity. So, the steps are as follows:

1. Design an architecture stack and decide on various libraries to use (let's say you choose Spring Framework for the backend, with Spring Security and Spring MVC).
2. Create an application base with all the technologies wired together (for example, you will have to make sure the authentication flow between the Angular client-side and Spring Security is wired properly).
3. Write a build system for the application (let's say you used webpack to build the Angular client-side and Gradle to build the server-side).
4. Write integration tests and unit tests for the base.
5. Create administrative modules.
6. Design business entities and create them with the Angular client-side and Java server-side with test coverage.
7. Write all the business logic.
8. Build Docker images, test the application, and deploy it.

While this approach definitely works, for this simple application you would have spent anywhere between 4 to 6 weeks depending on the team size. Now, more than 70% of the effort would have been spent on writing boilerplate code and making sure all the libraries work well together. Now, would you believe me if I say that you could develop, test, and deploy this application in less than 30 minutes using JHipster? Yes, you can, while still getting high-quality, production-grade code with lots of extra bells and whistles. We will see this in action in our next chapter, where we will build a real-world application using JHipster.

Let's take a look at the goal and adoption of JHipster.

## Goal and adoption of JHipster

The goal of JHipster is to provide developers with a platform where you can focus on your business logic rather than worrying about wiring different technologies together, and also one that provides a great developer experience. Of course, you can use available boilerplate within your organization or from the internet and try to wire it up together, but then you will be wasting a lot of time reinventing the wheel. With JHipster, you will create a modern web application or microservices architecture with all the required technologies wired together and working out of the box, such as the following:

- A robust and high-performance Spring Framework-based Java stack on the backend
- A rich mobile-first frontend with Angular, React, or Vue.js supported by Bootstrap
- A battle-tested microservices architecture unifying Netflix OSS, Elastic Stack, Docker, and Kubernetes
- A great tooling and development workflow using Maven/Gradle, webpack, and npm
- Out-of-the-box continuous integration using Jenkins, Travis, Azure DevOps, or GitLab
- Excellent Docker support and support for orchestration tools such as Kubernetes and OpenShift out of the box
- Out-of-the-box support for deployment of the application with major cloud providers
- Above all, great code with lots of best practices and industry standards at your fingertips



Netflix OSS (<https://github.com/Netflix>) is a collection of open source tools and software produced by the Netflix, Inc. team geared toward microservices architecture. In the current cloud-native world, these libraries and tools are being replaced by tools from the Kubernetes ecosystem.

JHipster has been steadily increasing in popularity as Spring Boot and Angular/React/Vue.js have gained momentum, and lots of developers have started to adopt them as the de facto frameworks for web development. As per official statistics at the time of writing (end of 2019), there are more than 20,000 applications generated per month and JHipster has been installed around 2.5 million times. It has more than 500 contributors, with official contributions from Google, Microsoft, RedHat, Heroku, and so on.



**Elastic Stack** (<https://www.elastic.co/products>), formerly known as **ELK Stack**, is a collection of software tools that help in the monitoring and analytics of microservices developed by the Elasticsearch team.

In the next section, we'll learn about the different technologies supported by JHipster.

## Introduction to available technologies

JHipster supports an incredible number of modern web application technologies out of the box. Some of them are used as the base or core of the generated application, while some technologies are opt-in via choices made during application generation. Let's see the different technologies supported mainly for monolithic applications in brief:

- Client-side technologies
- Server-side technologies
- Database options

There are many more technologies supported, and we will look at them in later chapters when we touch upon microservices.

Client-side technologies are explained in the following sections.

## Client-side technologies

The role of client-side technologies in full stack development has grown from just using JavaScript for client-side validations to writing full-blown, single-page applications using client-side MVVM frameworks. The frameworks and toolchains used have become complex and overwhelming for developers who are new to the client-side landscape. Fortunately for us, JHipster provides support for most of the following widely used client-side technologies. Let us take a brief look and get familiar with the important tools and technologies that we will use. No need to worry if it is overwhelming; we will take a deeper look at some of the more important ones during the course of the book.

In the next section, we'll learn about the different client-side technologies.

## HTML5 and CSS3

Web technologies, especially HTML and CSS, have undergone major updates and are becoming better day by day due to excellent support in modern browsers.

HTML5 (<https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5>) is the latest of the **HyperText Markup Language (HTML)** standard, which introduces new elements, attributes, and behaviors. The term is used to collectively refer to all the HTML technologies used to build modern web applications. This iteration introduced support for features such as offline storage, WebSockets, web workers and WebGL. JHipster also uses best practices from HTML5 Boilerplate (<https://html5boilerplate.com>).



**HTML5 Boilerplate** is a collection of modern technologies, default settings, and best practices that kick-start modern web development faster.

CSS3 (<https://developer.mozilla.org/en-US/docs/Web/CSS/CSS3>) is the latest of the **Cascading Style Sheets (CSS)** specification. It adds support for media query, animations, flexbox, round corners, and a lot more. CSS3 makes it possible to natively animate elements, apply special effects, apply filters, and so on to get rid of the many JavaScript hacks that were used earlier.



Flexible box, or flexbox, is a layout mode ([https://developer.mozilla.org/en-US/docs/Web/CSS/Layout\\_mode](https://developer.mozilla.org/en-US/docs/Web/CSS/Layout_mode)) that can be used instead of the box model used traditionally. This allows you to have a flexible box model, making responsive layouts easier to handle without floats and margin collapse issues.

## SASS

**Syntactically Awesome Style Sheets (SASS)** (<http://sass-lang.com>) is a CSS extension language. It is preprocessed and converted to CSS during compile time. It has similar semantics to CSS and is 100% compatible with all versions of CSS. It additionally supports advanced features such as nested syntax, variables, mixins, inheritance, partials, and so on. SASS makes it possible to reuse CSS and to write maintainable style sheets. JHipster uses SASS by default for its CSS3.

## Bootstrap

Bootstrap (<https://getbootstrap.com>) is a responsive UI framework for modern web development. It offers a mobile-first approach for web development with utilities and UI components that are fully responsive. Bootstrap 4 is the latest version, uses flexbox for layout, and is completely written in SASS, which makes it easy to customize. Bootstrap supports a 12-column grid framework, which lets you build responsive web pages with ease. JHipster uses ng-bootstrap (<https://ng-bootstrap.github.io>) for Angular and reactstrap (<https://reactstrap.github.io/>) for React so that pure Angular/React components are used instead of the ones provided by Bootstrap, which are built using jQuery, and Bootstrap is used only for styling.



**Mobile-first web development** is an approach where the UX/UI is designed for smaller screen sizes first, thus forcing you to focus on the most important data/elements to be presented. This design is then gradually enhanced for bigger screen sizes making the end result responsive and efficient.

JHipster also supports Bootswatch (<https://bootswatch.com/>) themes for Bootstrap. You can choose a theme from the supported Bootswatch themes while generating the application.

## MVVM framework

**Model-View-View-Model (MVVM)** is an architectural pattern originally developed by Microsoft. It helps to abstract or separate the client-side (GUI) development from the server-side (data model). The view model is an abstraction of the *View* and represents the state of data in the *Model*. With JHipster, you can choose between Angular, React, and Vue.js as the client-side framework.



There is also an official JHipster Vue.js blueprint (<https://github.com/jhipster/jhipster-vuejs>) if you would like to use Vue.js as the client-side framework. We will have a dedicated section regarding this later in the book.

We'll take a look at Angular, React, and Vue.js in the next sections.

## Angular

Angular (<https://angular.io>; version 2 and above) is a complete backward-incompatible rewrite of the original AngularJS framework) is a client-side MVVM framework, maintained by Google, which helps to develop **single-page applications (SPAs)**. It is based on a declarative programming model and it extends standard HTML with the ability to add additional behavior, elements, and attributes through components.

Angular is written in TypeScript and recommends the use of TypeScript to write Angular applications as well. Angular removed some of the concepts that were used in AngularJS, such as scope, controller, and factory. It also has a different syntax for binding attributes and events. Another major difference is that the Angular library is modular and hence you can choose the modules that you need, to reduce bundle size. Angular also introduced advanced concepts such as **Ahead-of-Time (AOT)** compilation, lazy loading and Reactive Programming.



TypeScript is a superset of ECMAScript 6 (ES6 – version 6 of JavaScript) and is backward-compatible with ES5. It has additional features such as static typing, generics, and class attribute visibility modifiers. Since TypeScript is a superset of ES6, we can also use ES6+ features (<http://es6-features.org>) such as modules, lambdas (arrow functions), generators, iterators, string templates, reflection, and spread operators.

## React

React (<https://reactjs.org>) is not a full-fledged MVVM framework. It is a JavaScript library for building client-side views or user interfaces. It is developed and backed by Facebook and has a vibrant community and ecosystem behind it. React follows an HTML in JavaScript approach and has a special format called JSX to help us write React components. Unlike Angular, React doesn't have too many concepts or APIs to learn and hence is easier to start with, but React only cares about rendering the UI and hence to get similar functionality offered by Angular, we would have to pair React with other libraries such as React Router (<https://reacttraining.com/react-router>), Redux (<https://redux.js.org>), and MobX (<https://mobx.js.org>). JHipster uses React along with Redux and React Router and similar to Angular, JHipster uses TypeScript to write React as well. But this is optional as React can be written using JavaScript as well, preferably ES6 (<http://es6-features.org>). React is fast to render due to its use of a virtual DOM (<https://reactjs.org/docs/faq-internals.html>) to manipulate a view instead of using the actual browser DOM.

## Vue.js

Vue.js (<https://vuejs.org>) is a progressive JavaScript framework. It is open source and completely community-driven. Vue.js is what a marriage between AngularJS and React would look like. It has a similar syntax to AngularJS but has the speed and performance of React. Like React, Vue.js is also a UI framework that can be combined with other libraries to build SPAs. Vue.js can be written in JavaScript or TypeScript and can be used to write small web components or full-fledged SPAs. JHipster has an official blueprint to provide Vue.js support.

## Build tools – webpack and BrowserSync

The client-side has evolved a lot and become as complex as the server-side, hence it requires a lot more tools in your toolbelt to produce optimized results. You need a build tool to transpile, minimize, and optimize your HTML, JavaScript, and CSS code. One of the most popular is webpack. JHipster uses webpack for Angular, React, and Vue.js.

Webpack (<https://webpack.js.org>) is a module bundler with a very flexible loader/plugin system. Webpack walks through the dependency graph and passes it through the configured loaders and plugins. With webpack, you can transpile TypeScript to JavaScript, minimize, and optimize CSS and JavaScript, compile SASS, revision, hash your assets, and so on. Webpack can remove dead code in a process called **tree shaking**, thus reducing bundle size. Webpack is configured using a configuration file and can be run from the command line or via NPM/Yarn scripts.

Then we have BrowserSync (<https://browsersync.io>), which is a Node.js tool that helps with browser testing by synchronizing file changes and interactions of the web page across multiple browsers and devices. It provides features such as auto-reload on file changes, synchronized UI interactions, synchronized scrolling, and so on. It integrates with webpack to provide a productive development setup. It makes testing a web page on multiple browsers and devices super easy.

## Testing tools – Jest and Protractor

Gone are the days when the client-side code didn't require unit testing. With the evolution of client-side frameworks, the testing possibilities also improved. There are many frameworks and tools available for unit testing, end-to-end testing, and more. JHipster creates unit tests for client-side code using Jest out of the box and also supports creating end-to-end tests using Protractor.

Jest (<https://jestjs.io/>) is a JavaScript testing framework. It can work with TypeScript, Angular, React, and Vue.js. It has a simple API and great features and integrates well with continuous integration tools.

Protractor (<http://www.protractortest.org>) is an end-to-end testing framework developed by the Angular team. It was originally intended for Angular and AngularJS applications but it is flexible enough to be used with any framework, such as React, jQuery, and Vue.js. Protractor runs end-to-end tests against real browsers using the Selenium web driver API.

## Internationalization

**Internationalization (i18n)** is a very important feature these days and JHipster supports this out of the box. Multiple languages can be chosen during application creation. On the client-side, this is achieved by storing GUI text in JSON files per language and using an Angular/React library to dynamically load this based on the language selected at runtime.



Do you know why internationalization is abbreviated as i18n? Because there are 18 characters between i and n. There are other similarly named abbreviations in web technology, for example, **accessibility (a11y)**, **localization (l10n)**, **globalization (g11n)**, and **localizability (l12y)**.

Server-side technologies are explained in the following sections.

## Server-side technologies

Server-side technologies in web development have evolved a lot, and with the rise of frameworks such as Spring and Play, the need for Java EE has reduced and opened doors for more feature-rich alternatives, such as Spring Boot. Some of the core technologies such as Hibernate are here to stay, while newer concepts such as JWT, Liquibase, Swagger, Kafka, and WebSockets bring a lot of additional opportunities. Let us take a quick look at some of the important technologies supported by JHipster; we will encounter these later on in the book and will take a deeper look at some of these technologies.

In the next section, we'll learn about different server-side technologies.

## Spring Framework

Spring Framework (<https://spring.io>) might be the best thing since sliced bread in the Java world. It changed the Java Web application landscape for the good. The landscape was monopolized by Java EE vendors before the rise of Spring and soon after Spring, it became the number one choice for Java Web developers, giving Java EE a run for its money. At its core, Spring is an **Inversion of Control (IoC)** (<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans>) container providing dependency injection and application context. The main features of Spring or the Spring triangle, combine IoC, **Aspect-Oriented Programming (AOP)** (<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop>), and technology abstractions in a consistent way. The framework has numerous modules aimed at different tasks, such as data management, security, REST, and web services. Spring Framework and its modules are free and open source. Let us see some of the important modules in a bit more detail.



IoC is a software design pattern where custom or task-specific code is invoked by a library, rather than the traditional procedural programming approach where custom code calls libraries when required. IoC helps to make the code more modular and extendable. AOP provides another way of thinking about program structure. The unit of modularity is the aspect that enables the modularization of concerns such as transaction management that cut across multiple types and objects.

We'll take a look at Spring Boot, Spring Security, Spring MVC, and Spring Data in the next sections.

## Spring Boot

Spring Boot (<https://spring.io/projects/spring-boot>) is a widely used solution these days for Java web application development. It has an opinionated convention over configuration approach. It is completely configuration driven and makes using Spring Framework and many other third-party libraries a pleasure. Spring Boot applications are production-grade and can just run in any environment that has a JVM installed. It uses an embedded servlet container such as Tomcat, Jetty, or Undertow to run the application. It autoconfigures Spring wherever possible with sensible defaults and has starter POM for many modules and third-party libraries. It does not require any XML configuration and lets you customize autoconfigured beans using Java configuration.



JHipster, by default, uses Undertow as the embedded server in the applications generated. Undertow is very lightweight and faster to start and is ideal for the development and production of lightweight applications.

## Spring Security

Spring Security (<https://spring.io/projects/spring-security>) is the de facto solution for security in a Spring Framework-based application. It provides an API and utilities to manage all aspects of security, such as authentication and authorization. It supports a wide range of authentication mechanisms, such as OAuth2, JWT, Session (web form), LDAP, SSO (short for **single sign-on**) servers, JAAS (short for **Java Authentication and Authorization Service**), and Kerberos. It also has features such as remember me and concurrent sessions.

## Spring MVC

Spring MVC (<https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html>) is the default solution to work with the servlet API within Spring applications. It is a request-based system and abstracts the servlet API to make it easier to design controllers to serve HTTP requests. REST is the de facto standard for designing API endpoints these days, and Spring MVC REST is a specific subset that makes it easier to design and implement RESTful services.

## Spring Data

Spring Data (<https://spring.io/projects/spring-data>) is a module that abstracts data access operations for many different data access technologies and databases. It provides a consistent API to work seamlessly with different underlying implementations. This frees us from worrying about the underlying database and data access technology. It has powerful features such as dynamic query generation from method names and custom object mapping abstractions. Spring data supports working with JPA, MongoDB, Redis, and Elasticsearch, to name a few. It also lets you export Spring Data repositories as RESTful resources.

## Security

In modern web applications, there are multiple ways to implement authentication and authorization. Spring Security supports a wide range of mechanisms, as we saw earlier, and JHipster provides support for the following standards.

We'll take a look at JWT, Session, and OAuth2/OIDC in the next sections.

### JWT

**JSON Web Token (JWT)** (<https://jwt.io>) is an open industry standard for security tokens. JWT authentication works by a server and client passing and verifying claims. A server generates a JWT token and passes it back to the client when user credentials are successfully validated. The client will store this token locally and use it to request protected resources from the server later by passing the token in the request header. This is a stateless authentication mechanism. This is explained in detail in Chapter 8, *Microservice Server-Side Technologies*.

### Session

Session-based authentication is the traditional web form-based authentication mechanism where the server creates and maintains a session for the validated user credentials. This is stateful and normally is not very scalable unless you use a distributed HTTP session, which is possible using a distributed cache such as Hazelcast or using the session replication features of a dedicated web server or load balancer. JHipster adds a lot of features on top of the standard mechanism, such as secured tokens that are stored in the database, sessions can be invalidated, remember me mechanisms, and so on.

### OAuth2/OIDC

**OAuth2** (<https://developer.okta.com/blog/2017/06/21/what-the-heck-is-oauth>) is a protocol for stateless authentication and authorization. The protocol allows applications to obtain limited access to user accounts on services. User authentication is delegated to service, typically, an OAuth2 server. OAuth2 is more complicated to set up when compared to the previously mentioned mechanisms.



**OpenID Connect** (<https://openid.net/connect/>) is an identity layer on top of the OAuth 2.0 protocol. It is governed by the OpenID foundation and has a specification of the implementation and hence allows for implementation by different vendors.

JHipster supports setting up OAuth2 with **OpenID Connect (OIDC)** and can use Keycloak (<https://keycloak.org>) or Okta (<https://developer.okta.com/blog/2017/10/20/oidc-with-jhipster>) out of the box.

## Build tools – Maven and Gradle

JHipster supports using either Maven or Gradle as the build tool for the server-side code. Both are free and open source.

**Maven** (<https://maven.apache.org>) is a build automation tool that uses an XML document called `pom.xml` to specify how an application is built and its dependencies. Plugins and dependencies are downloaded from a central server and cached locally. The Maven build file is called a **Project Object Model (POM)** and it describes the build process. Maven has a long history and is much more stable and reliable than Gradle. It also has a huge ecosystem of plugins. JHipster provides its own **BOM** (short for **Bill Of Materials**; <https://github.com/jhipster/jhipster>) to make dependency management easier.

**Gradle** (<https://gradle.org>) is a build automation tool that uses a Groovy DSL to specify the build plan and dependencies. Gradle is much more flexible and feature-rich than Maven, making it an ideal choice for very complex build setups. The latest version of Gradle easily surpasses Maven in terms of speed and features. Another unique advantage of Gradle is the ability to write standard Groovy code in the build script, making it possible to do pretty much everything programmatically. It has great plugin support as well. Gradle also provides a Kotlin DSL (<https://gradle.org/kotlin/>).

## Hibernate

Hibernate (<http://hibernate.org>) is the most popular **ORM** (short for **object-relational mapping**) tool for Java. It helps to map an object-oriented domain model to a relational database scheme using Java annotations. It implements **JPA** (short for **Java Persistence API**) and is the go-to provider for a JPA implementation. Hibernate also offers many additional features, such as entity auditing and bean validation. Hibernate automatically generates SQL queries depending on the underlying database semantics and makes it possible to switch the databases of an application very easily. It also makes the application database independent without any vendor lock-in. Hibernate is free and open source software.

## Liquibase

Liquibase (<http://www.liquibase.org>) is a free and open source version control tool for the database. It lets you track, manage, and apply database schema changes using configuration files without having to fiddle with SQL. It is database-independent and goes well with JPA, making the application database independent. Liquibase can be run from within the application, making database setup and management seamless. Liquibase can also add/remove data to/from a database, making it good for migrations as well.

## Caching

Caching is a good practice in software development, and it improves the performance of read operations considerably. Caching can be enabled for Hibernate second level cache, and also with Spring Cache abstraction, to enable caching at the method level.



Spring Cache abstraction (<https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#cache>) lets us easily add caching at different layers of the application. For example, with this, we can enable caching of a pure function (which doesn't involve any side effects) so that for the same set of inputs, the outputs are produced from the cache instead of executing the method.

JHipster supports JCache-compatible Hibernate second level cache provided by Ehcache, Hazelcast, Redis, and Infinispan.

We'll take a look at Ehcache, Hazelcast, Infinispan, Memcached, and Redis in the next sections.

### Ehcache

Ehcache (<http://www.ehcache.org>) is an open source JCache provider and is one of the most widely used Java caching solutions. It is JCache compatible and is a good choice for applications that are not clustered. For clustered environments, additional Terracotta servers (this is an in-memory data platform providing both caching and operational storage) are required. It is stable, fast, and simple to set up.

## Hazelcast

Hazelcast (<https://hazelcast.org>) is an open source distributed in-memory data grid solution. It has excellent support for clustered applications and distributed environments and hence is a good choice for caching. While Hazelcast has numerous other features and use cases, caching remains one of the important ones. It is highly scalable and a good option for microservices due to its distributed nature.

## Infinispan

Infinispan (<http://infinispan.org>) is a distributed cache and key-value store from Red Hat. It is free and open source. It supports clustered environments and is hence a good choice for microservices. It has features such as in-memory data grids and MapReduce support.

## Memcached

Memcached (<https://memcached.org/>) is a high-performance distributed in-memory object cache system. It's a simple but effective cache solution. It can be used with Spring cache abstraction at the moment in JHipster but not as a second level Hibernate cache.

## Redis

Redis (<https://redis.io/>) is an open source in-memory data store. It has features such as replication, Lua scripting, LRU eviction, transactions, and different levels of on-disk persistence. High availability can be configured using Redis Sentinel and automatic partitioning can be done with Redis Cluster. It can be used as a database, cache, and message broker. It can be used in a JHipster application via the Spring Cache Abstraction and as Hibernate second level cache.

## Swagger

The **OpenAPI specification** (previously known as the **Swagger specification**) is an open standard for designing and consuming RESTful web services and API. The OpenAPI specification is a standard founded by a variety of companies, including Google, Microsoft, and IBM. The Swagger (<https://swagger.io>) name is now used for the associated tooling. JHipster supports the API-first development model with Swagger code-gen and also supports API visualization with Swagger UI.

## Thymeleaf

Thymeleaf (<http://www.thymeleaf.org>) is an open source Java server-side templating engine with very good integration with Spring. Thymeleaf can be used to generate web pages on the server-side, for templating email messages, and more. Although server-side web page templates are slowly losing out to client-side MVVM frameworks, it is still a useful tool if you want to have something more than a SPA using Angular.

## Micrometer

Micrometer metrics (<https://micrometer.io/>) is an open source vendor-neutral facade for application metrics on JVM. Paired with Spring Boot, this can provide a lot of value by measuring the performance of the REST API, measuring the performance of the cache layer and database, and more. Micrometer provides a handy list of annotations to mark methods to be monitored. It integrates with popular monitoring solutions.

## WebSocket

WebSocket ([https://developer.mozilla.org/en-US/docs/Web/API/WebSockets\\_API](https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API)) is a communication protocol that works on top of TCP. It provides a full-duplex communication channel over a single TCP connection. It was standardized by W3C (<https://www.w3.org>). It is lightweight and enables real-time communication between a client and a server. In terms of web applications, this enables the server to communicate with the client app in the browser without a request from the client. This opens the door to push data from server to client in real time and for implementations such as real-time chat and notifications. On the server-side, JHipster relies on Spring, which provides the necessary support (<https://spring.io/guides/gs/messaging-stomp-websocket/>) to work with WebSocket.

## Kafka

Kafka (<https://kafka.apache.org>) is an open source stream processing system. It has a distributed pub/sub-based message queue for storage. Its fault tolerance and scaling capabilities have helped it to replace JMS and AMQP as the preferred messaging queue. Spring provides an abstraction on top of Kafka to make it easier to configure and work with Kafka.



JMS (short for **Java Message Service**) is a messaging standard developed for Java EE and enables sending and receiving asynchronous messages between components using topics and queues. AMQP (short for **Advanced Message Queuing Protocol**; <https://www.amqp.org/>) is an open standard protocol for message-oriented middleware, providing features such as queuing, routing, and publish-subscribe mechanisms.

## Testing frameworks – JUnit, Gatling, and Cucumber

Server-side testing can be mainly categorized into unit testing, integration testing, performance testing, and behavior testing. JHipster supports all of these with JUnit, Gatling, and Cucumber, out of which JUnit comes out of the box, while the others are opt-in.

**JUnit** (<https://junit.org/junit5/>) is the most widely used Java testing framework. It is free and open source software. It was originally intended for unit testing but combined with Spring Test Framework (<https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#testing-introduction>), it can also be used for integration testing. JHipster creates unit tests and REST API integration tests using JUnit and Spring Test Framework.

**Gatling** (<https://gatling.io/>) is a free and open source performance and load testing tool. It is based on Scala and uses a Scala DSL to write test specs. It creates detailed reports of the load testing and it can be used to simulate all kinds of load on a system. It is a required tool for performance-critical applications.

**Cucumber** (<https://cucumber.io/>) is a **behavior-driven development (BDD)** testing framework used mainly for acceptance testing. It uses a language parser called Gherkin, which is very human-readable as it looks similar to plain English.

With this, we have covered both the client- and server-side technologies currently supported by JHipster. Now let's discuss the available database options.

## Introduction to database options

Today, there are a wide variety of database options out there. These can be broadly classified into the following:

- SQL databases
- NoSQL databases



You can visit <https://db-engines.com/en/ranking> to see the popularity of different databases.

JHipster supports some of the most widely used databases, as detailed here.

## SQL databases

SQL databases or **relational database management systems (RDBMSes)** are those that support a relational table-oriented data model. They support table schema defined by the fixed name and number of columns/attributes with a fixed data type. Each row in a table contains a value for every column. Tables can be related to each other.

We'll take a look at H2, MySQL, MariaDB, PostgreSQL, MS SQL, and Oracle in the next sections.

### H2

H2 (<http://www.h2database.com/html/main.html>) is a free embedded RDBMS that's commonly used for development and testing. It can normally run in filesystem mode for persistence or in-memory mode. It has a very small footprint and is extremely easy to configure and use. It doesn't have many of the enterprise features offered by other mainstream database engines and hence is normally not preferred for production usage.

### MySQL

MySQL (<https://www.mysql.com/>) is one of the most popular database engines and is free and open source software. It is from Oracle but also has a very vibrant community. It has enterprise-ready features such as sharding, replication, and partitioning. It is one of the most popular SQL databases these days.

### MariaDB

MariaDB (<https://mariadb.org/>) is a MySQL-compliant database engine with an additional focus on security, performance, and high availability. It is gaining popularity and is sought after as a good alternative for MySQL. It is free and open source software.

## PostgreSQL

PostgreSQL (<https://www.postgresql.org/>) is another free and open source database system that is very much in demand. It is actively maintained by a community. One of the unique features of PostgreSQL is the advanced JSON object storage with the capability to index and query within the JSON. This makes it possible to also use it as a NoSQL database or in Hybrid mode. It also has enterprise-ready features such as replication and high availability.

## MS SQL

MS SQL Server (<https://www.microsoft.com/en-us/sql-server/default.aspx>) is an enterprise database system developed and supported by Microsoft. It is commercial software and requires a paid license to use. It has enterprise-ready features and premium support from Microsoft. It is one of the popular choices for mission-critical systems.

## Oracle

Oracle (<https://www.oracle.com/database/index.html>) is the most widely used database due to its legacy and enterprise features. It is a commercial software and requires a paid license to use. It has enterprise-ready features such as sharding, replication, and high availability.

## NoSQL databases

This is a wide umbrella that encompasses any database that is not an RDBMS. This includes document stores, wide column stores, search engines, key-value stores, graph DBMS, and content stores. A general trait of such databases is that they can be schema-less and do not rely on relational data.

## MongoDB

MongoDB (<https://www.mongodb.com/>) is a cross-platform document store and is one of the most popular choices for NoSQL databases. It has a proprietary JSON-based API and query language. It supports MapReduce and enterprise features such as sharding and replication. It is a free and an open source software.



**MapReduce** is a data processing paradigm where a job is split into multiple parallel map tasks, with the produced output sorted and reduced into the result. This makes processing large datasets efficient and fast.

## Cassandra

Apache Cassandra (<http://cassandra.apache.org/>) is a distributed column store with a focus on high availability, scalability, and performance. Due to its distributed nature, it doesn't have a single point of failure, making it the most popular choice for critical high-availability systems. It was originally developed and open sourced by Facebook.

Did you know? Cassandra can have up to 2 billion columns per row!



## Couchbase

Couchbase (<https://www.couchbase.com/>) is a commercially supported NoSQL database. It is a distributed document-oriented database and has enterprise-grade features and support.

## Elasticsearch

Elasticsearch (<https://www.elastic.co/products/elasticsearch>) is a search and analytics engine based on Apache Lucene (<http://lucene.apache.org/>). It is technically a NoSQL database, but it is primarily used as a search engine due to its indexing capability and high performance. It can be distributed and multi-tenant with full-text search capability. It has a web interface and JSON documents. It is one of the most widely used search engines.

Now we're ready to install and set up JHipster on your system.

# Installing and setting up JHipster

To get started with JHipster, you will have to install the JHipster CLI tool. The JHipster CLI comes with commands required to use all of the features offered by the platform.



**JHipster Online:** If you would like to create an application without installing anything, you can do so by visiting <https://start.jhipster.tech>. You can authorize the application to generate a project directly in your GitHub account or you can download the source as a ZIP file.

Before we install the JHipster CLI, let's take a look at the prerequisites.

## Prerequisites

We will need to install some dependencies and configure our favorite IDE to work best with the generated code. You can visit <http://www.jhipster.tech/installation/> to get up-to-date information about this.

## Tools required

The following are the tools required to install JHipster and to work with the generated applications. If you do not have them installed already, perform the following steps and install them.

You will need to use a command-line interface (Command Prompt or Terminal application) throughout this section, and hence it is better to have one open. Since the installation of some of the following tools will alter the environment variables, you might have to close and reopen the Terminal after the installation of a tool:

- On Windows, use the default Command Prompt (cmd) or PowerShell.
- On Linux, use Bash or your favorite Terminal emulator.
- On macOS, use iTerm or your favorite Terminal application.

## Java

JHipster supports the latest LTS version of Java (Java 11). While JHipster applications will work with Java versions 8 to 13, it is recommended to stick to the latest stable LTS release (Java 11).

The generated applications use Java 8 features to be backward compatible, and hence Java 8 is the minimum required version to compile the applications:

1. Check for your installed Java version by running the `java -version` command in the Terminal. It should display Java version `x.x.x`, where `x.x.x` could be any version between 1.8 and 12.0.
2. If you do not have the correct version installed, you can visit the Oracle website (<http://www.oracle.com/technetwork/java/javase/downloads/index.html>) and follow the instructions to install the JDK for Java 11.
3. Once the JDK is installed, check the command in *step 1* again to make sure. As the JDK alters the environment variable to set `JAVA_HOME`, you will have to open a new Terminal here.



Alternatively, you can also use SDKMAN (<https://sdkman.io/>) to manage multiple Java versions from different vendors. Follow the installation (<https://sdkman.io/install>) and usage instructions (<https://sdkman.io/usage>) from the website.

## Git

Git is the most widely used version control system for source code management. It promotes distributed revision control and is an integral part of development these days.

JHipster uses Git for upgrading applications, and Git is also recommended for the smooth working of Node.js and npm ecosystems:

1. Check for Git by running `git --version` in the Terminal. It should display Git version `x.x.x`; the version number could be anything.
2. If the command is not found, you can visit `git-scm` (<https://git-scm.com/downloads>) and follow the instructions to install Git on your operating system.
3. Once installed, run the command in *step 1* again to make sure.

## Node.js

Node.js is a JavaScript runtime environment. It revolutionized the JavaScript world and made JavaScript the most popular development language among developers today (according to <https://insights.stackoverflow.com/survey/2019#most-popular-technologies>). The Node ecosystem is the largest in the world, with over 800,000 packages (<http://www.modulecounts.com/>), and is managed by `npm`, the default package manager.

The JHipster CLI is a Node.js application and hence requires Node.js to run, and many of the tools used in the generated application will also require Node.js:

1. Check for Node.js by typing `node -v` in the Terminal. It should display a version number. Make sure that the version number is greater than 8.10 and corresponds to the latest LTS version of Node.js.
2. If the command is not found or if you have a lower version of Node.js then you can visit the Node.js website (<https://nodejs.org/en/download/>) and follow the instructions to install the latest LTS version. Please note that non-LTS versions (current) might not be stable, and it is advised not to use them.
3. Once installed, check the command in *step 1* again to make sure. As Node.js alters the environment variables, you will have to open a new Terminal here.
4. `npm` is automatically installed when you install Node.js. You can check this by running `npm -v` in the Terminal.



JHipster uses NPM by default, but if you prefer to use Yarn, that is possible as well. You can visit the Yarn website (<https://yarnpkg.com/en/docs/install>) and follow the instructions to install Yarn.

## Docker

**Docker** is the de facto standard for container management, and it makes using containers a breeze. It provides tools to create, share, and deploy containers.

You will need Docker and `docker-compose` to run the generated database images and for the development of microservices:

1. Check for Docker by running `docker -v` in a Terminal. It should display a version number.
2. Check for Docker Compose by running `docker-compose -v` in a Terminal. It should display a version number. If you are on Mac or Linux, you could just run `docker -v && docker-compose -v` together.
3. If the command is not found, you can visit the Docker website (<https://docs.docker.com/install/>) and follow the instructions to install it. Also, install Docker Compose (<https://docs.docker.com/compose/install/>) by following the instructions.
4. Once installed, check the command in *step 1* again to make sure.



Optionally, install a Java build tool. Normally, JHipster will automatically install the Maven Wrapper (<https://github.com/takari/maven-wrapper>) or the Gradle Wrapper ([https://docs.gradle.org/current/userguide/gradle\\_wrapper.html](https://docs.gradle.org/current/userguide/gradle_wrapper.html)) for you, based on your choice of the build tool. If you don't want to use those wrappers, go to the official Maven website (<http://maven.apache.org/>) or the Gradle website (<https://gradle.org/>) to do your own installation.

## IDE configuration

JHipster applications can be created by using a command-line interface and JHipster CLI. Technically speaking, an IDE is not a requirement, but when you continue the development of a generated application, it is highly recommended that you use a proper Java IDE such as IntelliJ, Eclipse, or NetBeans. Sometimes you can also use advanced text editors such as Visual Studio Code or Atom with appropriate plugins to get the work done. Depending on the IDE/text editor you choose, it is recommended to use the following plugins to make development more productive:

- **Angular/React:** TSLint, TypeScript, editor config
- **Java:** Spring, Gradle/Maven, Java language support (VS Code)

Regardless of IDE/text editor, always exclude the `node_modules`, `git`, `build`, and `target` folders to speed up indexing. Some IDEs will do this automatically based on the `.gitignore` file.



Visit <http://www.jhipster.tech/configuring-ide/> in your favorite browser to read more about this.

## System setup

Before installing and diving into JHipster, here are a few pointers to prepare you for some of the common issues that you might encounter:

- If you are using Yarn on macOS or Linux, you need to have `$HOME/.config/yarn/global/node_modules/.bin` in the path. This will normally be automatically done when you install Yarn, but if not, you can run the `export PATH="$PATH:`yarn global bin`:$HOME/.config/yarn/global/node_modules/.bin"` command in a Terminal to do this.
- If you are behind a corporate proxy, you will have to bypass it for npm, Git, and Maven/Gradle to work properly. Visit <http://www.jhipster.tech/configuring-a-corporate-proxy/> to see what proxy options can be set for different tools used.



If you are on Mac or Linux and if you are using Oh My Zsh or the Fisherman shell, then you could use the specific plugins from JHipster for that. Visit <http://www.jhipster.tech/shell-plugins/> for details.

OK, now let's get started for real. The next section explains the installation of JHipster.

## Installation of JHipster

JHipster can be used from a local installation with npm or Yarn or using a Docker image. Alternatively, there is also the JHipster online application we saw earlier.

Among all the options, the best way to utilize the full power of JHipster is by installing the JHipster CLI using NPM. Open a Terminal and run the following:

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

Wait for the installation to finish and, in the Terminal, run `jhipster --version`. You should see the version info, as shown here:

```
~
> jhipster --version
INFO! Using JHipster version installed globally
6.5.1
```

That's it; we are ready to roll.

If you are someone who cannot wait for new versions to arrive, you can always use the current development code by following these steps after installing the JHipster CLI:

1. In a Terminal, navigate to a directory you would like to use. For example, if you have a folder called `project` in your home directory, run `cd ~/projects/` and for Windows, run `cd c:\Users\<username>\Desktop\projects`.
2. Run `git clone https://github.com/jhipster/generator-jhipster.git`.
3. Now, navigate to the folder by running `cd generator-jhipster`.
4. Run `npm link` to create a symbolic link from this folder into the globally installed application in `global node_modules`.

5. Now when you run the JHipster commands, you will be using the version you cloned instead of the version you installed.
6. Once you generate an application, run `npm link generator-jhipster` in the application folder as well to get the global version we have linked.

Please note that you should be doing this only if you are absolutely sure of what you are doing. Also please note that development versions of the software will always be unstable and might contain bugs.

If you prefer to isolate the installation in a virtual environment, then you can use the Docker image from the JHipster team. Visit <http://www.jhipster.tech/installation> and scroll down to the **Docker installation (for advanced users only)** section for instructions on how to use a Docker image.



You can install multiple `npm` packages by running the `npm -g install webpack generator-jhipster` CLI command.

At the time of writing, the latest JHipster version is 6.5, and this will be used to build applications throughout the book.

## Summary

In this chapter, we discovered JHipster and the different technology options provided by it. We had a brief introduction to the important pieces of the client-side and server-side stack. We had a quick overview of Spring technologies, Angular, and Bootstrap. We also had an overview of different database options supported by JHipster. We learned about the tools required to work with JHipster, we successfully set up our environment to work with JHipster, and we installed JHipster CLI.

In the next chapter, we will see how JHipster can be used to build a production-grade monolithic web application.

# 2

## Section 2: Building and Customizing Web Applications with JHipster

In this section, we will add to what we learned from the previous section on JHipster and build monolithic web applications. You will also model and create entities using **JHipster Domain Language (JDL)** and JDL-Studio. In the last chapter of this section, you will learn how to develop further the applications generated using Angular, Bootstrap, Spring Security, Spring MVC REST, and Spring Data.

This section comprises the following chapters:

- Chapter 3, *Building Monolithic Web Applications with JHipster*
- Chapter 4, *Entity Modeling with JHipster Domain Language*
- Chapter 5, *Customization and Further Development*

# 3

# Building Monolithic Web Applications with JHipster

Let's get into action and build a production-grade web application using JHipster. Before we start, we need a use case. We will be building an e-commerce web application that manages products, customers, and their orders and invoices. The web application will use a MySQL database for production and will have an Angular frontend. The UI for the actual shopping website will be different from the back office features, which will only be available for employees who have an administrator role. In this exercise, we will only be building a simple UI for the client-facing part.

In this chapter, we will cover the following topics:

- How to create a monolithic web application using JHipster
- Important aspects of the code we've generated
- Security aspects of the application we've generated
- Run applications and tests
- Frontend screens we've generated
- Included tools that will ease further development



This chapter will require the use of a Terminal (Command Prompt on Windows) app throughout. Please read the previous chapter for more information about that.

# Application generation

Before we start generating our application, we need to prepare our workspace as this workspace will be used throughout this book; you will be creating many Git branches on this workspace as we proceed.



Visit <http://rogerdudler.github.io/git-guide/> for a quick reference guide on Git commands.

## Step 1 – preparing the workspace

Let's create a new folder for the workspace. Create a folder called `e-commerce-app` and, from the Terminal, navigate to the folder:

```
> mkdir e-commerce-app  
> cd e-commerce-app
```

Now, create a new folder for our application; let's call it `online-store` and navigate to it:

```
> mkdir online-store  
> cd online-store
```

Now, we are ready to invoke JHipster. First, we need to make sure everything is ready by running the `jhipster --version` command. It should print a globally installed JHipster version; otherwise, you'll need to follow the instructions from the previous chapter to set it up.



It is always better to use the latest versions of any tools as they might include important bug fixes. You can upgrade JHipster anytime using the `npm install -g generator-jhipster` or `yarn global upgrade generator-jhipster` commands.

## Step 2 – generating code using JHipster

Initialize JHipster by running the `jhipster` command in the Terminal, which will produce the following output:

```
> jhipster
INFO! Using JHipster version installed globally
INFO! Running default command
INFO! Executing jhipster:app
INFO! Options: from-cli: true


https://www.jhipster.tech

Welcome to JHipster v6.5.1
Application files will be generated in folder: /home/depu/Documents/jhipster-book/v2/online-store-monolith/e-commerce-app/online-store

Documentation for creating an application is at https://www.jhipster.tech/creating-an-app/
If you find JHipster useful, consider sponsoring the project at https://opencollective.com/generator-jhipster

? Which *type* of application would you like to create? (Use arrow keys)
❯ Monolithic application (recommended for simple projects)
  Microservice application
  Microservice gateway
  JHipster UAA server (for microservice OAuth2 authentication)
  [Alpha] Reactive monolithic application
  [Alpha] Reactive microservice application
```

JHipster will ask a number of questions to get input about different options that are required. The first question is about the application type that we want. Here, we are presented with the following options:

- **Monolithic application:** As the name suggests, it creates a monolithic web application with a Spring Boot-based backend and a SPA frontend.
- **Microservice application:** This creates a Spring Boot microservice without any frontend and is designed to work with a JHipster microservice architecture.
- **Microservice gateway:** This creates a Spring Boot application very similar to the monolithic application but geared toward a microservice architecture with additional configurations. It features a SPA frontend.
- **JHipster UAA server:** This creates an OAuth2 user authentication and authorization service. This will not feature any frontend code and is designed to be used in a JHipster microservice architecture.
- **[Alpha] Reactive monolithic application:** As the name suggests, it creates a reactive monolithic web application with a Spring Boot-based backend and a SPA frontend. Please note that this is currently an alpha feature.
- **[Alpha] Reactive microservice application:** This creates a Spring Boot reactive microservice without any frontend and is designed to work with a JHipster microservice architecture. Please note that this is currently an alpha feature.

We will choose the **Monolithic application** option for our use case. We will discuss microservice options in detail in Chapter 8, *Microservice Server-Side Technologies*.



Run `jhipster --help` to see all the available commands. Run `jhipster <command> --help` to see help information for a specific command; for example, `jhipster app --help` will display help information for the main app generation process.

In the following sections, we'll learn about server-side options.

## Server-side options

The generator will now start asking us about the server-side options that we need. Let's go through them one by one:

- **Question 1:** This prompt asks for a base name for the application, which is used to create the main class filenames, database names, and so on. By default, JHipster will suggest the current directory name if it doesn't contain any special characters. Let's name our application `store`. Please note that the files will be created in the directory you are currently in:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store_
```

- **Question 2:** This prompt asks for a Java package name. Let's choose `com.mycompany.store`:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store_
```

- **Question 3.** This prompt asks whether we need to configure the JHipster Registry for this instance. The JHipster registry provides service discovery and config server implementation, which is very useful for centralized configuration management and scaling the application. For this use case, we won't need it, so let's choose `No`. We will learn more about the JHipster Registry in Chapter 8, *Microservice Server-Side Technologies*:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? (Use arrow keys)
❯ No
Yes
```

- **Question 4:** This prompt asks us to select an authentication mechanism. Here, we are presented with three options:
  - JWT authentication
  - HTTP Session Authentication
  - OAuth 2.0 / OIDC Authentication

We already saw how these differ in the previous chapter. For our use case, let's choose JWT authentication:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? (Use arrow keys)
❯ JWT authentication (stateless, with a token)
HTTP Session Authentication (stateful, default Spring Security mechanism)
OAuth 2.0 / OIDC Authentication (stateful, works with Keycloak and Okta)
```

- **Question 5:** This prompt asks us to select a database type; the options that are provided are SQL, MongoDB, Couchbase, and Cassandra. You can also choose to have no database. We learned about various database options in the previous chapter. For our application, we'll choose a SQL database:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? (Use arrow keys)
❯ SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
MongoDB
Cassandra
Couchbase
No database
```

- **Question 6:** This prompt asks us to choose a specific SQL database that we would like to use in production; the available options are MySQL, MariaDB, PostgreSQL, Oracle, and Microsoft SQL Server. Let's choose MySQL:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? (Use arrow keys)
❯ MySQL
  MariaDB
  PostgreSQL
  Oracle (Please follow our documentation to use the Oracle proprietary driver)
  Microsoft SQL Server
```

- **Question 7:** This prompt asks us to choose between our chosen SQL database and the H2 embedded database for development. The H2 embedded DB is especially useful as it makes development faster and self-contained, without the need to have a MySQL instance running. Let's choose the H2 with disk-based persistence option here since it's lightweight and easier to use in development compared to having a full-fledged DB service running:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? MySQL
? Which *development* database would you like to use? (Use arrow keys)
❯ H2 with disk-based persistence
  H2 with in-memory persistence
  MySQL
```

If your use case requires working with persisted data in development and if the model isn't going to change often, then you could also choose **MySQL** for development as it would give you a faster startup time. This is because the embedded H2 DB doesn't need to be initialized. However, the downside is that each time you make schema changes or recreate entities, you would have to update the DB using generated Liquibase diff changelogs manually or wipe the DB manually and start over again. With an embedded H2 DB, you could run `./gradlew clean` to wipe it.



- **Question 8:** This prompt asks us to choose a Spring cache implementation. We can choose between no-cache, Ehcache, Caffeine, Hazelcast, Infinispan, Redis, and Memcached. Since we learned about these in the previous chapter, let's go ahead and choose Hazelcast here:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? MySQL
? Which *development* database would you like to use? H2 with disk-based persistence
? Do you want to use the Spring cache abstraction?
  Yes, with the Ehcache implementation (local cache, for a single node)
  Yes, with the Caffeine implementation (local cache, for a single node)
> Yes, with the Hazelcast implementation (distributed cache, for multiple nodes, supports rate-limiting for gateway applications) [BETA]
  Yes, with the Infinispan implementation (hybrid cache, for multiple nodes)
  Yes, with Memcached (distributed cache) - Warning, when using an SQL database, this will disable the Hibernate 2nd level cache!
  Yes, with the Redis implementation (single server)
  No - Warning, when using an SQL database, this will disable the Hibernate 2nd level cache!
```

- **Question 9:** This prompt asks us to choose if we need a second level cache for Hibernate. Let's choose Yes. It will use the same cache implementation we chose for the previous question:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? MySQL
? Which *development* database would you like to use? H2 with disk-based persistence
? Do you want to use the Spring cache abstraction? Yes, with the Hazelcast implementation (distributed cache, for multiple nodes)
? Do you want to use Hibernate 2nd level cache? (Y/n) y_
```

- **Question 10:** This prompt let's us choose a build tool to use for the project; the options are Maven and Gradle. Let's choose Gradle here since it is more modern and powerful:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? MySQL
? Which *development* database would you like to use? H2 with disk-based persistence
? Do you want to use the Spring cache abstraction? Yes, with the Hazelcast implementation (distributed cache, for multiple nodes)
? Do you want to use Hibernate 2nd level cache? Yes
? Would you like to use Maven or Gradle for building the backend?
  Maven
> Gradle
```

- **Question 11:** This prompt is interesting as it presents various additional options supported by JHipster. The options are as follows:
  - **Elasticsearch:** Adds Elasticsearch support for the generated entities
  - **WebSockets:** Adds WebSocket support using Spring WebSocket, SocketJS, and the Stomp protocol
  - **API first development with OpenAPI-generator:** Adds OpenAPI generator support for API first development
  - **Apache Kafka:** Adds support for an asynchronous queue using Kafka

Let's keep it simple and choose WebSockets using Spring Websocket:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? MySQL
? Which *development* database would you like to use? H2 with disk-based persistence
? Do you want to use the Spring cache abstraction? Yes, with the Hazelcast implementation (distributed cache, for multiple nodes)
? Do you want to use Hibernate 2nd level cache? Yes
? Would you like to use Maven or Gradle for building the backend? Gradle
? Which other technologies would you like to use?
 Search engine using Elasticsearch
 WebSockets using Spring Websocket
 Asynchronous messages using Apache Kafka
 API first development using OpenAPI-generator
```

Now, we'll learn about client-side options.

## Client-side options

Now the generator will ask us about client-side options, including the client-side framework we wish to use:

- **Question 1:** This prompt asks us to select a client-side MVVM framework; the options include Angular and React. You can also choose to skip the client-side by selecting No client. Let's choose Angular:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? MySQL
? Which *development* database would you like to use? H2 with disk-based persistence
? Do you want to use the Spring cache abstraction? Yes, with the Hazelcast implementation (distributed cache, for multiple nodes)
? Do you want to use Hibernate 2nd level cache? Yes
? Would you like to use Maven or Gradle for building the backend? Gradle
? Which other technologies would you like to use? WebSockets using Spring Websocket
? Which *Framework* would you like to use for the client? (Use arrow keys)
❯ Angular
  React
  No client
```

- **Question 2.** This prompt lets us choose a Bootswatch (<https://bootswatch.com/>) theme for our application. Let's choose the default theme:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? MySQL
? Which *development* database would you like to use? H2 with disk-based persistence
? Do you want to use the Spring cache abstraction? Yes, with the Hazelcast implementation (distributed cache, for multiple nodes)
? Do you want to use Hibernate 2nd level cache? Yes
? Would you like to use Maven or Gradle for building the backend? Gradle
? Which other technologies would you like to use? WebSockets using Spring Websocket
? Which *Framework* would you like to use for the client? Angular
? Would you like to use a Bootswatch theme (https://bootswatch.com/)? (Use arrow keys)
❯ Default JHipster
  Cerulean
  Cosmo
  Cyborg
  Darkly
  Flatly
  Journal
(Move up and down to reveal more choices)
```

We'll learn about internationalization options in the next section.

## Internationalization options

Now we have the opportunity to enable internationalization and select the languages we would like to use:

- **Question 1.** This prompt lets us enable **internationalization (i18n)**. Let's choose yes:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? MySQL
? Which *development* database would you like to use? H2 with disk-based persistence
? Do you want to use the Spring cache abstraction? Yes, with the Hazelcast implementation (distributed cache, for multiple nodes)
? Do you want to use Hibernate 2nd level cache? Yes
? Would you like to use Maven or Gradle for building the backend? Gradle
? Which other technologies would you like to use? WebSockets using Spring WebSocket
? Which *framework* would you like to use for the client? Angular
? Would you like to use a Bootswatch theme (https://bootswatch.com/)? Default JHipster
? Would you like to enable internationalization support? (Y/n) y_
```

- **Question 2:** Since we enabled i18n, we will be given the option to choose a primary language and additional i18n languages. At the time of writing, there are 44 supported languages, including 2 **Right to Left (RTL)** languages. Let's choose English as the primary language and Chinese (Simplified) as an additional language:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? MySQL
? Which *development* database would you like to use? H2 with disk-based persistence
? Do you want to use the Spring cache abstraction? Yes, with the Hazelcast implementation (distributed cache, for multiple nodes)
? Do you want to use Hibernate 2nd level cache? Yes
? Would you like to use Maven or Gradle for building the backend? Gradle
? Which other technologies would you like to use? WebSockets using Spring WebSocket
? Which *framework* would you like to use for the client? Angular
? Would you like to use a Bootswatch theme (https://bootswatch.com/)? Default JHipster
? Would you like to enable internationalization support? Yes
? Please choose the native language of the application English
? Please choose additional languages to install (Press <space> to select, <a> to toggle all, <i> to invert selection)
 Belarusian
 Bengali
 Catalan
 Chinese (Simplified)
 Chinese (Traditional)
 Czech
 Danish
(Move up and down to reveal more choices)
```

In the next section, we'll learn about testing options.

## Testing

Now we can choose testing options for our application.

This prompt lets us choose testing frameworks for our application, which will also create sample tests for the application and entities. The options are Gatling, Cucumber, and Protractor. Let's choose Protractor:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? MySQL
? Which *development* database would you like to use? H2 with disk-based persistence
? Do you want to use the Spring cache abstraction? Yes, with the Hazelcast implementation (distributed cache, for multiple nodes)
? Do you want to use Hibernate 2nd level cache? Yes
? Would you like to use Maven or Gradle for building the backend? Gradle
? Which other technologies would you like to use? WebSockets using Spring Websocket
? Which *Framework* would you like to use for the client? Angular
? Would you like to use a Bootswatch theme (https://bootswatch.com/)? Default JHipster
? Would you like to enable internationalization support? Yes
? Please choose the native language of the application English
? Please choose additional languages to install Chinese (Simplified)
? Besides JUnit and Jest, which testing frameworks would you like to use?
 Gatling
 Cucumber
 Protractor
```

We'll learn about available modules in the next section.

## Modules

This prompt lets us choose additional third-party modules from the JHipster Marketplace (<https://www.jhipster.tech/modules/marketplace>). This can be helpful if we want to use additional features that aren't supported directly by JHipster. We will look at this in later chapters. For now, let's choose no. Don't worry about this; these modules can also be added to the application later when required:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? store
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? MySQL
? Which *development* database would you like to use? H2 with disk-based persistence
? Do you want to use the Spring cache abstraction? Yes, with the Hazelcast implementation (distributed cache, for multiple nodes)
? Do you want to use Hibernate 2nd level cache? Yes
? Would you like to use Maven or Gradle for building the backend? Gradle
? Which other technologies would you like to use? WebSockets using Spring Websocket
? Which *Framework* would you like to use for the client? Angular
? Would you like to use a Bootswatch theme (https://bootswatch.com/)? Default JHipster
? Would you like to enable internationalization support? Yes
? Please choose the native language of the application English
? Please choose additional languages to install Chinese (Simplified)
? Besides JUnit and Jest, which testing frameworks would you like to use? Protractor
? Would you like to install other generators from the JHipster Marketplace? (y/N) n_
```

Once all the questions have been answered, the code generation process will start and you will see an output, listing the files that were created and then running the NPM installation to get all the frontend dependencies installed. The generator might present additional prompts, where the Angular team asks you to approve the process of collecting usage statistics. This is up to you.



If you do not want the NPM install and webpack build steps to run, you could use the `--skip-install` flag while running JHipster to skip these. Just run `jhipster --skip-install`.

The installation process could take up to a few minutes, depending on your network speed and system configuration.



JHipster will check your environment to see if all the required dependencies, such as Java 8, NodeJS, Git, and NPM/Yarn, are installed. If not, it will show friendly warning messages before code generation starts.

Once the process is complete, you will see messages indicating it's been successful and instructions to start the application:

```
Application successfully committed to Git from /home/deepu/Documents/jhipster-book/v2/online-store-monolith/e-commerce-app/online-store.
If you find JHipster useful consider sponsoring the project https://www.jhipster.tech/sponsors/
Server application generated successfully.

Run your Spring Boot application:
./gradlew

Client application generated successfully.

Start your Webpack development server with:
npm start

> store@0.0.1-SNAPSHOT cleanup /home/deepu/Documents/jhipster-book/v2/online-store-monolith/e-commerce-app/online-store
> rimraf build/resources/main/static/ build/resources/main/aot

INFO! Congratulations, JHipster execution is complete!
```



There are also command-line flags that can be passed while executing the `jhipster` command. Running `jhipster app --help` will list all available command-line flags. One interesting flag, for example, is `yarn`, which lets you use Yarn instead of NPM for dependency management.

JHipster will automatically initialize a Git repository for the folder and commit the generated files. If you wish to do this step yourself, you can do so by providing the `skip-git` flag during execution. Run `jhipster --skip-git` and execute these steps manually as follows:

```
> git init
> git add --all
> git commit -am "generated online store application"
```



You could also use a GUI tool such as Sourcetree (<https://www.sourcetreeapp.com/>) or GitKraken (<https://www.gitkraken.com/>) if you wish to do so to work with Git.

Now, let's take a look at our application code and explore the important parts to understand the application better.

## Code walkthrough

Now that we have generated our application with JHipster, let's go through some important pieces of the source code that has been created. Let's open our application in our favorite IDE or Editor.

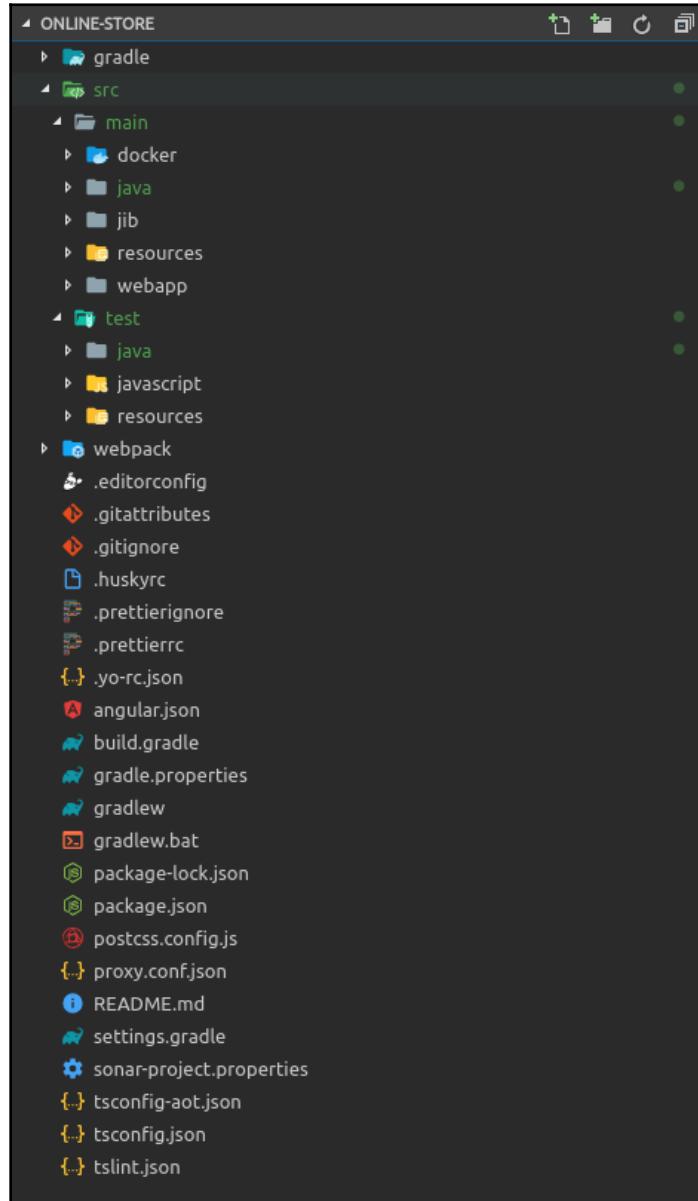


If you are using IntelliJ IDEA, you can execute `idea .` in a Terminal from the application folder to launch it. Otherwise, you can import the application as a new Gradle project using the **File | New | Project** menu option from existing sources; select the project folder before selecting **Gradle** from the options, clicking **Next**, and then clicking **Finish**. If you are using Eclipse, open the **File | Import...** dialog, select **Gradle Project** from the list, and follow the instructions shown. If you prefer to use VSCode, you can run `code .` from the application folder to launch it.

Let's start off by taking a look at the file structure of the application we created.

## File structure

Once created, the application will have the following file structure:



As you can see, the root folder is quite busy since it contains a few folders but a lot of configuration files. The most interesting among them are as follows:

- `src`: This is the source folder which holds the main application source and the test source files.
- `webpack`: This folder holds all the webpack client-side build configurations for development, production, and testing.
- `gradle`: This folder contains a Gradle wrapper and additional Gradle build scripts that will be used by the main Gradle build file (JHipster provides a similar wrapper if Maven is chosen as well).
- `build.gradle`: This is our Gradle build file and specifies our application's build life cycle. It also has specifies server-side dependencies. The build uses the properties defined in the `gradle.properties` file alongside it. You'll also find an executable named `gradlew` (`gradlew.bat` for Windows), which lets you use Gradle without having to install it.
- `.yo-rc.json`: This is the configuration file for JHipster. This file stores the options we selected during app creation, and it is used for app regeneration and upgrades.
- `package.json`: This NPM configuration file specifies all your client-side dependencies, client-side build dependencies, and tasks.
- `tsconfig.json`: This is the configuration for Typescript. There is also `tsconfig-aot.json` for Angular **ahead-of-time (AOT)** compilation.
- `.eslintrc.json`: This is the lint configuration for the application.



Install and configure Typescript and the ESLint plugin for your IDE or editor to make the most of Typescript.

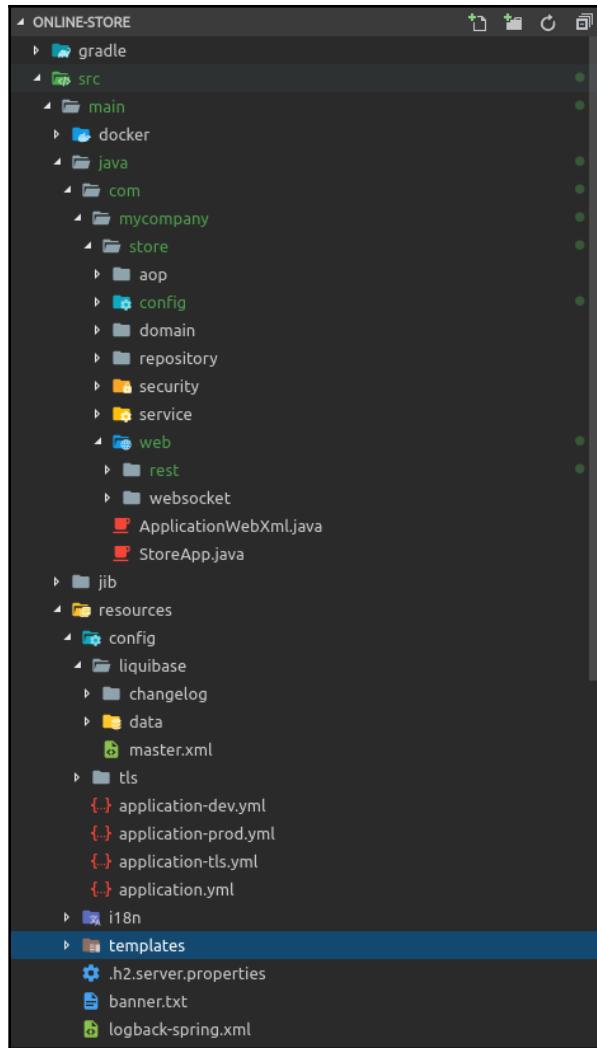
Now, let's take a look at the source folder. The `main` folder will hold the main app source code and the `test` folder will **test** the source code. The folder structure is as follows:

- `main`:
  - `docker`: Holds the Dockerfile for the application and the Docker compose configurations for selected options
  - `java`: Holds the main Java source code for the application
  - `resources`: Holds Spring Boot configuration files, Liquibase changelogs, and static resources such as server-side i18n files and email templates that are used by the application
  - `webapp`: Holds the Angular application source code and the client-side static content such as images, stylesheets, i18n files, and so on
- `test`:
  - `java`: Holds the unit and integration test source for the server-side
  - `javascript`: Holds the Karma unit test specs and Protractor end-to-end specs for the client-side application
  - `resources`: Holds Spring configuration files and static resources such as server-side i18n files and email templates that are used by the application to perform tests

We'll take a look at the server-side source code in the next section.

## Server-side source code

The server-side code is situated in the `java` and `resources` folder under `src/main`, as shown in the preceding screenshot. The folder's structure is as follows:



You may notice that Spring components do not use the traditional `@Autowired` or `@Inject` annotations for dependency injection in generated code. This is because we use constructor injection instead of field injection, and Spring Boot doesn't need explicit annotations for constructor injection. Constructor injection is considered better as it enables us to write better unit tests and avoids design issues, whereas field injection is more elegant but easily makes a class monolithic. The Spring team recommends constructor injection as a best practice. Constructor injection also makes unit-testing components easier.



We'll take a look at the Java source code that's generated in the next section.

## Java source

The important parts of the Java source code are as follows:

- `StoreApp.java`: This is the main entry class for the application. Since this is a Spring Boot application, the main class is executable and you can start the application by running this class from an IDE. Let's take a look at this class:
  - The class is annotated with a bunch of Spring JavaConfig annotations:

```
@SpringBootApplication
@EnableConfigurationProperties({LiquibaseProperties.class,
    ApplicationProperties.class})
```

- The first one, `@SpringBootApplication`, allows the required configurations for Spring to scan the source files and auto-detects Spring components (services, repository, resource, configuration classes that define Spring beans, and so on). It also tries to guess and auto-configure beans that the application might need based on classes that are found on the classpath and the configurations we have provided.
- The second one, `@EnableConfigurationProperties`, helps us register additional configurations for the application via property files.
- The main method of the class bootstraps the Spring Boot application and runs it:

```
public static void main(String[] args) {
    SpringApplication app = new
        SpringApplication(StoreApp.class);
    DefaultProfileUtil.addDefaultProfile(app);
    Environment env = app.run(args).getEnvironment();
    logApplicationStartup(env);
}
```

- `config`: This package contains Spring bean configurations for the database, cache, WebSocket, and so on. This is where we will configure various options for the application. Some important ones include:
  - `CacheConfiguration.java`: This class configures the Hazelcast cache for the application.

- `DatabaseConfiguration.java`: This class configures the database for the application and enables transaction management, JPA auditing, and JPA repositories for `application.SecurityConfiguration.java`: This is a very important part of the application as it configures security for it. Let's take a look at the important parts of the class:

- Following annotations enable web security and method-level security so that we can use the `@Secured` and `@Pre/PostAuthorize` annotations on individual methods:

```
@EnableWebSecurity  
@EnableGlobalMethodSecurity(prePostEnabled = true,  
    securedEnabled = true)
```

- The following configuration tells the application to ignore static content and certain APIs from the Spring Security configuration:

```
public void configure(WebSecurity web) {  
    web.ignoring()  
        .antMatchers(HttpMethod.OPTIONS,  
            "/**")  
        .antMatchers("/app/**/  
            .{js,html}")  
        .antMatchers("/i18n/**")  
        .antMatchers("/content/**")  
        .antMatchers("/h2-console/**")  
        .antMatchers("/swagger-ui  
            /index.html")  
        .antMatchers("/test/**");  
}
```

- The following configuration tells Spring Security which endpoints are permitted for all users, which endpoints should be authenticated, and which endpoints require a specific role (ADMIN, in this case):

```
public void configure(HttpSecurity http) throws Exception {  
    // @formatter:off  
    http  
        .csrf()  
        .disable()  
        .addFilterBefore(corsFilter,  
            UsernamePasswordAuthentication  
            Filter.class)  
        .exceptionHandling()  
        .authenticationEntryPoint  
            (problemSupport)  
        .accessDeniedHandler(problem
```

```
        .Support)
        .and()
        .headers()
        .contentSecurityPolicy("...")
        .and()
        .referrerPolicy(...)

        .and()
        .featurePolicy("...")
        .and()
        .frameOptions()
        .deny()
        .and()
        .sessionManagement()
        .sessionCreationPolicy(Session
            CreationPolicy.STATELESS)
        .and()
        .authorizeRequests()
        .antMatchers("/api/authenticate")
        .permitAll()
        .antMatchers("/api/register").
        permitAll()
        .antMatchers("/api/activate").
        permitAll()
        .antMatchers("/api/account/reset-
            password/init").permitAll()
        .antMatchers("/api/account/reset-
            password/finish").permitAll()
        .antMatchers("/api/**").
        authenticated()
        .antMatchers("/websocket/tracker")
        .hasAuthority(Authorities
            Constants.ADMIN)
        .antMatchers("/websocket/**").
        permitAll()
        .antMatchers("/management
            /health").permitAll()
        .antMatchers("/management
            /info").permitAll()
        .antMatchers("/management
            /prometheus").permitAll()
        .antMatchers("/management/**").
        hasAuthority(Authorities
            Constants.ADMIN)
        .and()
        .httpBasic()
        .and()
        .apply(securityConfigurer
            Adapter());
```

```
// @formatter:on  
}
```

- `WebConfigurer.java`: This is where we set up HTTP cache headers, MIME mappings, static asset locations, and **Cross-Origin Resource Sharing (CORS)**.

JHipster provides great CORS support out of the box:



- CORS can be configured using the `jhipster.cors` property, as defined in the JHipster common application properties (<http://www.jhipster.tech/common-application-properties/>).
  - CORS support is enabled by default in `dev` mode for monoliths and gateways. It is disabled by default for microservices as you are supposed to access them through a gateway.
  - CORS support is disabled by default in `prod` mode for both monoliths and microservices, for security reasons.
- 
- `domain`: Domain model classes for the application are in this package. These are simple **POJOs** (short for **Plain Old Java Objects**) which have JPA annotations mapping it to a Hibernate entity. When the **Elasticsearch** option is selected, these also act as Document objects. Let's take a look at the `User.java` class:
    - An entity class is characterized by the following annotations.  
The `@Entity` annotation marks the class as a JPA entity.  
The `@Table` annotation maps the entity to a database table.  
The `@Cache` annotation enables second-level caching of the entity, and it also specifies a caching strategy:

```
@Entity  
@Table(name = "jhi_user")  
@Cache(usage =  
CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
```

- There are various annotations that are used at the field level in these classes. `@Id` marks the primary key for the entity. `@Column` maps a field to a database table column by the same name when no override is provided. `@NotNull`, `@Pattern`, and `@Size` are annotations that are used for validation. `@JsonIgnore` is used by Jackson to ignore fields when converting objects that are to be returned in REST API requests into JSON. This is especially useful with Hibernate as it avoids circular references between relationships, which create tons of SQL DB requests and failures:

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    @Pattern(regexp = Constants.LOGIN_REGEX)
    @Size(min = 1, max = 50)
    @Column(length = 50, unique = true, nullable = false)
    private String login;

    @JsonIgnore
    @NotNull
    @Size(min = 60, max = 60)
    @Column(name = "password_hash", length = 60, nullable =
        = false)
    private String password;
```

- Relationships between database tables are also mapped to entities using JPA annotations. In the following, for example, it maps a many-to-many relationship between a user and user authorities. It also specifies a join table to be used for the mapping:

```
    @JsonIgnore
    @ManyToMany
    @JoinTable(
        name = "jhi_user_authority",
        joinColumns = {@JoinColumn(name = "user_id",
            referencedColumnName = "id")},
        inverseJoinColumns = {@JoinColumn(name =
            "authority_name", referencedColumnName = "name")})
    @Cache(usage =
        CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
    @BatchSize(size = 20)
    private Set<Authority> authorities = new HashSet<>();
```

- **repository:** This package holds Spring Data repositories for entities. These are typically interface definitions that are automatically implemented by Spring Data. This removes the need for us to write any boilerplate implementations for the data access layer. Let's look at the `UserRepository.java` example:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    ...
    Optional<User> findOneByActivationKey(String activationKey);
    List<User> findAllByActivatedIsFalseAndActivationKey
        IsNotNullAndCreatedDateBefore(Instant date);
    Optional<User> findOneByResetKey(String resetKey);
    Optional<User> findOneByEmailIgnoreCase(String email);
    Optional<User> findOneByLogin(String login);
    @EntityGraph(attributePaths = "authorities")
    Optional<User> findOneWithAuthoritiesById(Long id);
    @EntityGraph(attributePaths = "authorities")
    @Cacheable(cacheNames = USERS_BY_LOGIN_CACHE)
    Optional<User> findOneWithAuthoritiesByLogin(String login);
    @EntityGraph(attributePaths = "authorities")
    @Cacheable(cacheNames = USERS_BY_EMAIL_CACHE)
    Optional<User> findOneWithAuthoritiesByEmail(String email);
    Page<User> findAllByLoginNot(Pageable pageable, String login);
}
```

From the preceding code block, we can notice the following:

- The `@Repository` annotation marks this as a Spring Data repository component.
- The interface extends `JpaRepository`, which lets it inherit all default CRUD operations, such as `findOne`, `findAll`, `save`, `count`, and `delete`.

- Custom methods are written as simple method definitions that follow Spring Data naming conventions so that the method name specifies the query to be generated. For example, `findOneByEmailIgnoreCase` generates a query equivalent of `SELECT * FROM user WHERE LOWER(email) = LOWER(:email)`.
- `security`: This package holds Spring Security-related components and utilities. Since we chose JWT as our authentication mechanism, it holds JWT-related classes such as `TokenProvider`, `JWTFilter`, and `JWTConfigurer` as well.
- `service`: This package holds the service layer, which consists of Spring service beans, DTOs, MapStruct DTO mappers, and service utilities.
- `web`: This package holds web resource classes, view models classes, and utility classes:
  - `rest`: This package holds Spring resource classes for the REST API. It also holds view model objects and utilities. Let's take a look at `UserResource.java`:
    - Resource classes are marked with the `@RestController` and `@RequestMapping("/api")` annotations from Spring. The latter specifies the base URL path for the controller so that all `<applicationContext>/api/*` requests are forwarded to this class.
    - Request methods are annotated with annotations according to their purpose. For example, the following marks the `createUser` method as a `PostMapping` for `"/users"`, which means all POST requests to `<applicationContext>/api/users` will be served by this method. The `@PreAuthorize` annotation restricts the method's access to the specified role:

```
@PostMapping("/users")
@PreAuthorize("hasRole(\"" +
    AuthoritiesConstants.ADMIN + "\")")
public ResponseEntity<User> createUser(@Valid
    @RequestBody UserDTO userDTO) throws
    URISyntaxException {
    ...
}
```

- **WebSocket**: This package holds WebSocket controllers and view models.



JHipster uses **Data Transfer Objects (DTOs)** and **View Models (VMs)** on the server-side. DTOs are used to transfer data from the service layer to and from the resource layer. They **break** Hibernate transactions and avoid further lazy loading from being triggered by the resource layer since all the mapping is done without any references to the entity objects being held. VMs are only used to display data on the web frontend and don't interact with the service layer.

We'll look at configurations that are used for the application in the next section.

## Resources

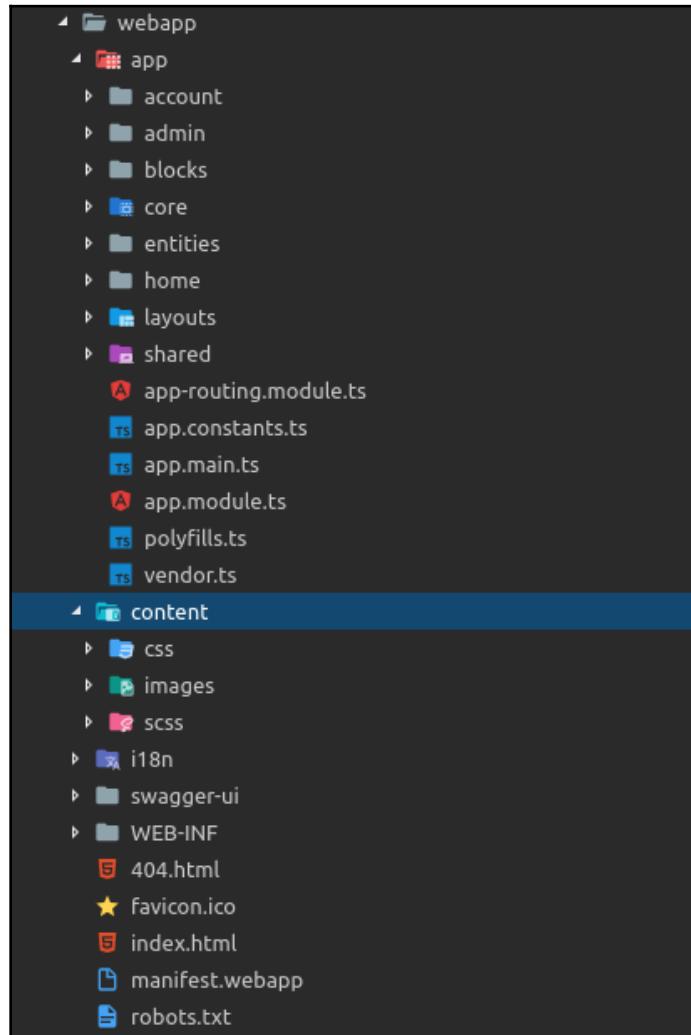
The important parts of resources are as follows:

- `config`: This holds application property YAML files, Liquibase changelogs, and the default key store that's used for the `tls` profile. The `application.yml` file holds configurable Spring Boot, JHipster, and application-specific properties, while `application.(dev|prod).yml` files hold properties that should be applied when the specific dev or prod profile is active. The test configurations are under `src/test/resource/application.yml`.
- `i18n`: This holds server-side i18n resource files.
- `templates`: This holds Thymeleaf templates for the client-side and emails.

We'll take a look at the client-side source code in the next section.

## Client-side source code

The client-side source code is in the `src/main/webapp` folder, as we saw earlier. The structure is as follows:



The most noteworthy pieces of information are as follows:

- `app`: This folder holds the Angular application's Typescript source code, which is organized with a folder per feature:

- `app.main.ts`: This is the main file for the Angular app. This bootstraps the Angular application. Notice that it uses `platformBrowserDynamic`, which lets the application work with **Just-in-Time (JIT)** compilation in the browser. This is ideal for development:

```
platformBrowserDynamic()
    .bootstrapModule(StoreAppModule, {
        preserveWhitespaces:
            true })
    .then(success => console.log(`Application started`))
    .catch(err => console.error(err));
```

- `app.module.ts`: This is the main module for the Angular app. It declares app-level components and imports other modules for the application. It also bootstraps the main application component:

```
@NgModule({
    imports: [
        BrowserModule,
        StoreSharedModule,
        StoreCoreModule,
        StoreHomeModule,
        // jhipster-needle-angular-add-module JHipster
        will
        // add new module here
        StoreEntityModule,
        StoreAppRoutingModule
    ],
    declarations: [JhiMainComponent, ...,
        FooterComponent],
    bootstrap: [JhiMainComponent]
})
export class StoreAppModule {}
```

- `account`: This module consists of account-related features such as `activate`, `password`, `password-reset`, `register`, and `settings`. Each typical component consists of `component.html`, `component.ts`, `route.ts`, and `service.ts` files.

- **admin:** This module consists of admin-related features such as audits, configuration, docs, health, logs, metrics, tracker, and user-management. Each typical component consists of `component.html`, `component.ts`, `route.ts`, and `service.ts` files.
- **core:** This module contains all core services (auth, tracker, user, login, language) and configures shared providers for the application.
- **blocks:** This folder consists of HTTP interceptors and other configs that are used by the application.
- **entities:** This is where entity modules will be created.
- **home:** The home page module.
- **layouts:** This folder has layout components such as the navbar, footer, error pages, and so on.
- **shared:** This module contains all shared components (login, auth, alert), entity models required for the application, and utilities required for the application.
- **content:** This folder contains static content such as images, CSS, and SASS files.
- **i18n:** This is where i18n JSON files live. Each language has a folder with numerous JSON files organized by modules.
- **swagger-ui:** This folder contains the Swagger UI client that's used in developing API documentation.
- **index.html:** This is the web application's index file. This contains very minimal code for loading the Angular application's main component. It is a single-page Angular application. You will also find a simple loading page and some commented out utility code such as a Google analytics script and service worker scripts on this file. These can be enabled if required:

```
<!doctype html>
<html class="no-js" lang="en" dir="ltr">
  <head>
    ...
  </head>
  <body>
    ...
    <jhi-main>
      ...
    </jhi-main>
    <noscript>
      <h1>You must enable javascript to view this page.</h1>
    </noscript>
```

```
...  
</body>  
</html>
```



To enable PWA mode using service workers, just uncomment the corresponding code in `src/main/webapp/index.html` to register the service worker. JHipster uses Workbox (<https://developers.google.com/web/tools/workbox/>), which creates the respective service worker and dynamically generates `service-worker.js`.

Now that we have learned everything about the application code let's start the application.

## Starting the application

Now let's start the application and view the output. There are multiple ways to run the application:

- By using the Spring Boot Gradle task from the Terminal/command line
- By executing the main Java class, `src/main/java/com/mycompany/store/StoreApp.java`, from an IDE
- By executing the packaged application file using the `java -jar` command

Let's start the application using the Gradle task. If you want to run the application directly in the IDE, just open the main app file that we mentioned earlier (`StoreApp.java`), right-click it, and choose **Run 'StoreApp'**.

To start the application via Gradle, open a Terminal/command line and navigate to the application folder. Then, execute the Gradle command as follows (if you are on Windows, execute `gradlew.bat`). This will trigger the default task, that is, `bootRun`:

```
> cd online-store  
> ./gradlew
```



Running `./gradlew` is equivalent to running `./gradlew bootRun -Pdev`. For the client-side, the webpack build task is run automatically, but if it fails for some reason, it can be triggered manually by running `npm run webpack:build`. This task can be triggered directly by the Gradle command as well by running `./gradlew webpack bootRun -Pdev`.

Gradle will start downloading the wrapper and dependencies. After a while, you should see console output somewhat similar to the following (in anywhere from a few seconds to a few minutes, depending on your network speed):

```
2017-10-15 22:23:53.512 INFO 7713 --- [ restartedMain] com.mycompany.store.StoreApp      : Started StoreApp in 15.134 seconds (JVM running for 16.204)
2017-10-15 22:23:53.513 INFO 7713 --- [ restartedMain] com.mycompany.store.StoreApp      :
-----
Application 'store' is running! Access URLs:
Local:          http://localhost:8080
External:        http://192.168.2.4:8080
Profile(s):     [swagger, dev]
-----
-> 90% EXECUTING [1m 49s]
```

The app has started successfully and is available on `http://localhost:8080`. Open your favorite browser and navigate to the URL.



Note that the preceding build will stay at 90% since the process is running continuously.

We'll learn about application modules in the next section.

## Application modules

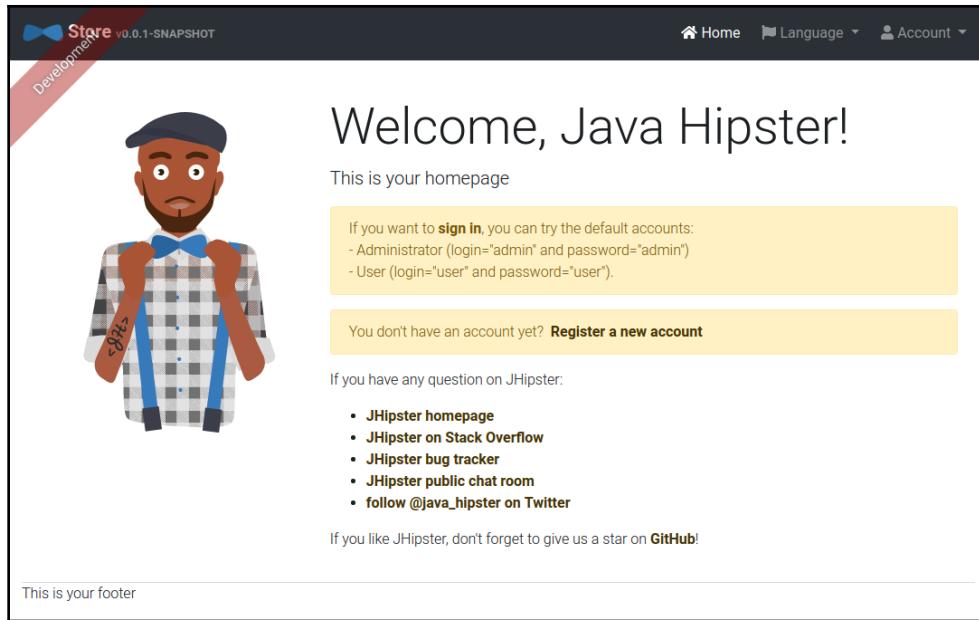
Let's look at the different modules that are available out of the box. These modules can be grouped as follows:

- Home and login
- Account
- Admin

We'll take a look at the different modules in the following sections.

## Home and login modules

Once you open the URL, you will see a cool-looking hipster on the home page, as follows:

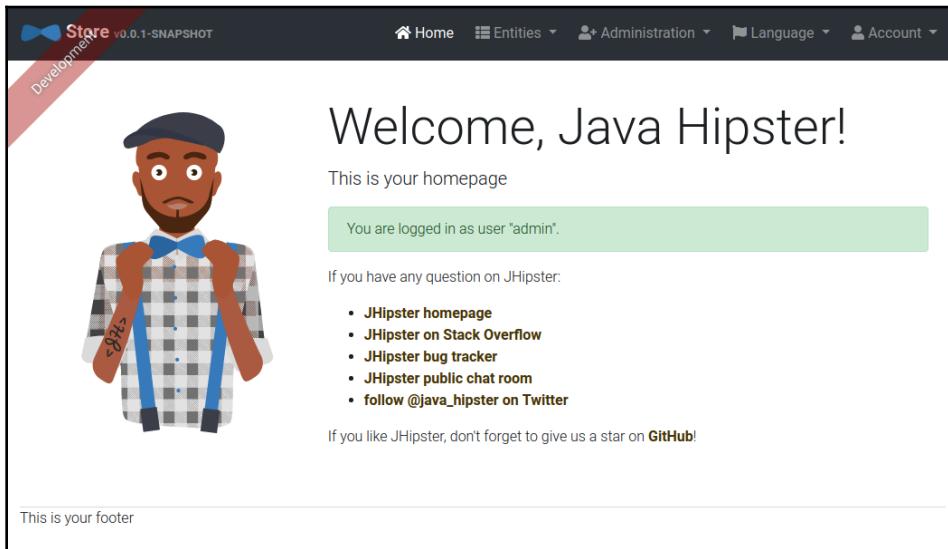


This is the home page. Let's log in to the application using the default credentials:

1. Click on the **Sign in** link on the page, or **Account | Sign in**. You will see the following login screen. Enter the default credentials, that is, **Username**—admin and **Password**—admin, and click **Sign in**:

A screenshot of the JHipster sign-in form. The title is "Sign in" with a close button "X". It has fields for "Username" (with "admin" entered) and "Password" (with "....." entered). There is a "Remember me" checkbox and a "Sign in" button. Below the form is a yellow callout box with the text "Did you forget your password?". At the bottom is another yellow callout box with the text "You don't have an account yet? **Register a new account**".

Once you've signed in, you will see the authenticated home page with all authenticated menu items in the navbar:



2. Since we enabled internationalization, we get a **Language** menu. Let's try to switch to a different language. Click on the **Language** menu and choose the next available language:



We'll learn about account modules in the next section.

## Account modules

Now, let's look at the account modules that are created out of the box. Under the **Account** menu, you will see a **Sign out** option and the following modules:

- Settings
- Password
- Registration

We'll take a look at the **Settings** module in the next section.

## Settings

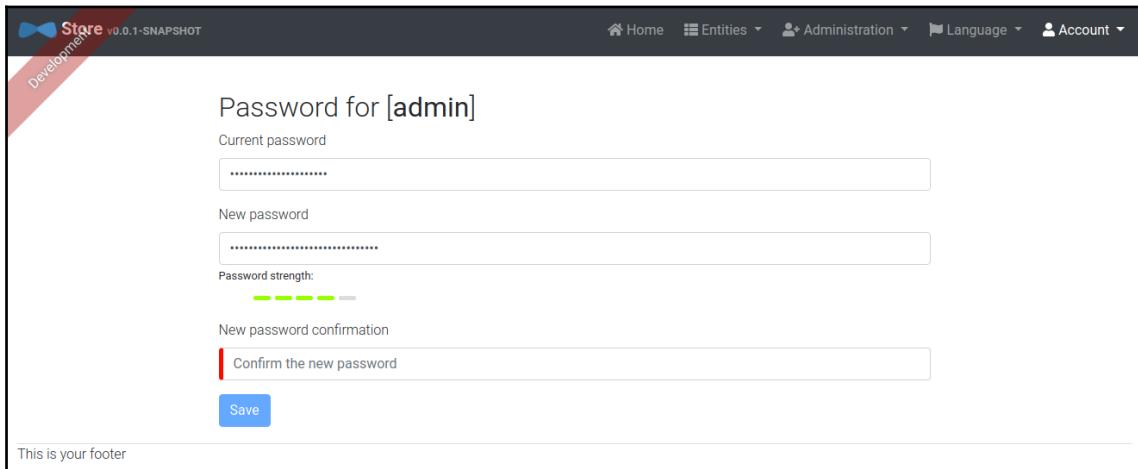
This module lets you change user settings such as name, email, and language:

The screenshot shows a web application interface for managing user settings. At the top, there is a navigation bar with links for Home, Entities, Administration, Language, and Account. A red diagonal banner on the left side of the header area says "Development". The main content area has a title "User settings for [admin]". It contains four input fields: "First Name" with value "Administrator", "Last Name" with value "Administrator", "Email" with value "admin@localhost", and a dropdown menu for "Language" set to "English". Below these fields is a blue "Save" button. At the bottom of the page, there is a footer note: "This is your footer".

We'll take a look at the **Password** module in the next section.

## Password

This module lets you change the password for the current user. There is also a forgot password flow with email verification out of the box:



The screenshot shows a JHipster application interface. At the top, there's a navigation bar with links for Home, Entities, Administration, Language, and Account. A red banner on the left says "Development" and "Store v0.0.1-SNAPSHOT". The main content area has a title "Password for [admin]". It contains four input fields: "Current password" (with placeholder "\*\*\*\*\*"), "New password" (with placeholder "\*\*\*\*\*"), "New password confirmation" (with placeholder "Confirm the new password" and a red border), and a "Save" button. Below the input fields is a "Password strength:" bar consisting of three green segments followed by two grey segments. At the bottom of the page, a footer bar contains the text "This is your footer".



To use the email features, you will have to configure an SMTP server in the application properties. We will look at this in [Chapter 15, Best Practices with JHipster](#).

We'll take a look at the **Registration** module in the next section.

## Registration

This module is only available when you are not logged in. This lets you sign up/register as a new user for the application. This will trigger a user activation flow with an activation email and verification process. This module will not be available if you choose **OAuth2** as your authentication method:

The screenshot shows the registration form for the JHipster Store application. The top navigation bar includes links for Home, Language, and Account. A red diagonal banner on the left says "Development". The main title is "Registration". The form fields are:

- Username: A text input field containing "Your username". Below it is an error message: "Your username is required."
- Email: A text input field containing "Your email".
- New password: A text input field containing "New password".
- Password strength: A progress bar consisting of five grey dashes.
- New password confirmation: A text input field containing "Confirm the new password".

A blue "Register" button is located below the input fields. At the bottom, a yellow callout box contains text: "If you want to **sign in**, you can try the default accounts:  
- Administrator (login="admin" and password="admin")  
- User (login="user" and password="user")."

We'll learn about the admin module in the next section.

## Admin module

Now, let's look at the **Admin** module screens that are generated. These are very useful for monitoring and developing the application. Under the **Admin** menu, you will find the following modules:

- User management
- Metrics
- Health
- Configuration
- Audits
- Logs
- API

We'll learn about the different admin modules in the following sections.

## User management

This module provides you with CRUD functionality so that you can manage users. The results are paginated by default. By default, users who register using the registration module will be deactivated unless they complete the registration process:

The screenshot shows the 'Users' page of the JHipster application. At the top, there's a navigation bar with links for Home, Entities, Administration, Language, and Account. A red banner on the left says 'Development Store v0.0.1-SNAPSHOT'. On the right, there's a blue button '+ Create a new user'. The main area has a table with columns: ID, Login, Email, Status, Language, Profiles, Created date, Modified by, and Modified date. The table contains three rows:

ID	Login	Email	Status	Language	Profiles	Created date	Modified by	Modified date
1	system	system@localhost	Activated	en	ROLE_USER ROLE_ADMIN		system	
3	admin	admin@localhost	Activated	en	ROLE_USER ROLE_ADMIN		system	
4	user	user@localhost	Activated	en	ROLE_USER		system	

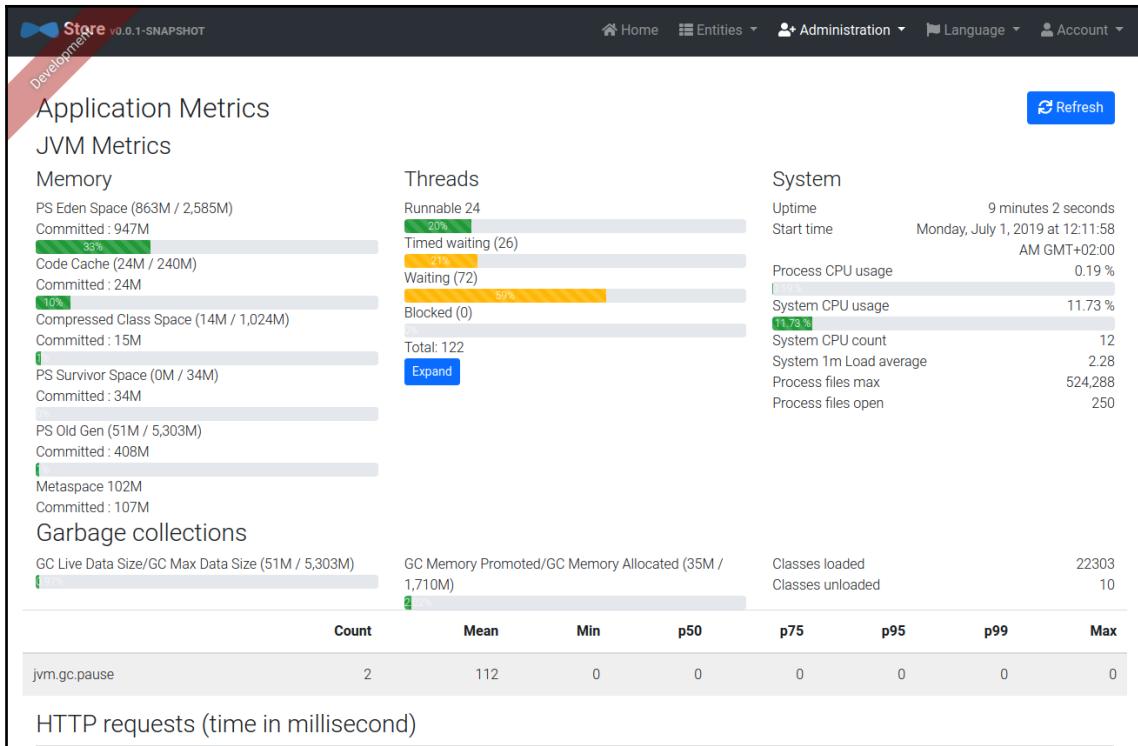
Below the table, it says 'Showing 1 - 3 of 3 items.' with a pagination control showing page 1 of 1.

At the bottom left, there's a footer message: 'This is your footer'.

We'll take a look at the **Metrics** module in the next section.

## Metrics

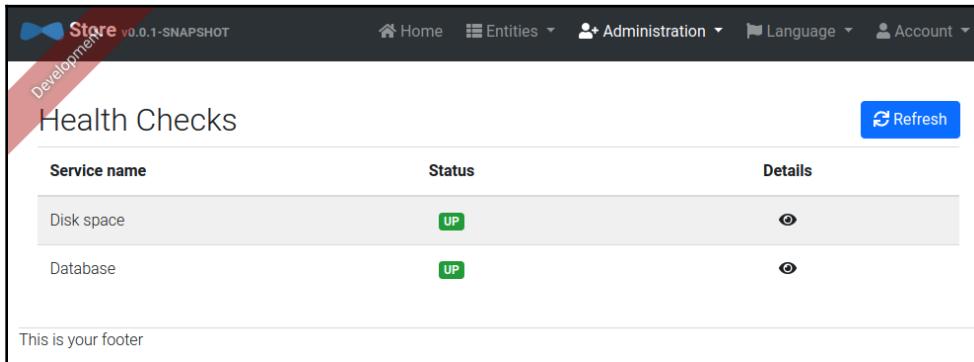
This module visualizes data provided by the Spring Boot actuator and Dropwizard metrics. This is very useful for monitoring application performance as it gives method-level performance information along with JVM, HTTP, database, and cache metrics. The **Expand** button near **Threads** will let you see the thread dump as well:



We'll take a look at the **Health** module in the next section.

## Health

This module provides the health status of application components such as **Database** and other information such as **Disk space**:



Let's take a look at the **Configuration** module in the next section.

## Configuration

This module helps us visualize the current application configuration. This is very useful for troubleshooting configuration issues:

The screenshot shows the JHipster Configuration module interface. At the top, there is a navigation bar with links for Home, Entities, Administration, Language, and Account. A red ribbon banner on the left says "Development". The main area is titled "Configuration" and includes a "Filter (by prefix)" input field. Below it, there are sections for "Spring configuration", "server.ports", "servletContextInitParams", and "systemProperties". The "systemProperties" section contains a table with the following data:

Property	Value
java.runtime.name	OpenJDK Runtime Environment
sun.boot.library.path	/home/deepu/.sdkman/candidates/java/8.0.212-zulu/jre/lib/amd64
java.vm.version	25.212-b04
java.vm.vendor	Azul Systems, Inc.
java.vendor.url	http://www.azulsystems.com/
path.separator	<input type="text"/>
java.vm.name	OpenJDK 64-Bit Server VM
file.encoding.pkg	sun.io

We'll take a look at the **Audits** module in the next section.

## Audits

This module lists all user authentication audit logs since JHipster enables audits for Spring Security, which means all security events are captured. There is a special Spring Data repository that writes audit events to the database. This is very useful from a security standpoint:

The screenshot shows the JHipster Store application interface. At the top, there's a navigation bar with links for Home, Entities, Administration, Language, and Account. A red ribbon banner on the left says "Development". The main content area has a title "Audits". Below it is a "Filter per date" section with input fields for "from" (01/06/2019) and "to" (02/07/2019). A table lists audit entries:

Date	User	State	Extra data
Jul 1, 2019, 12:20:08 AM	admin	AUTHENTICATION_SUCCESS	
Jul 1, 2019, 12:14:04 AM	admin	AUTHENTICATION_SUCCESS	

Below the table, a message says "Showing 1 - 2 of 2 items." with a page navigation bar showing pages 1, 2, and 3. At the bottom, a footer message reads "This is your footer".

We'll take a look at the **Logs** module in the next section.

## Logs

This module helps us view and update application log levels at runtime. This is very useful for troubleshooting:

The screenshot shows the JHipster Store application interface. At the top, there's a navigation bar with links for Home, Entities, Administration, Language, and Account. A red ribbon banner on the left says "Development". The main content area has a title "Logs". It displays a message "There are 1442 loggers." and a "Filter" input field. Below is a table listing loggers with their current log levels:

Name	Level
LiquibaseSchemaResolver	TRACE DEBUG INFO WARN ERROR OFF
ROOT	TRACE DEBUG INFO WARN ERROR OFF
ch	TRACE DEBUG INFO WARN ERROR OFF
ch.qos	TRACE DEBUG INFO WARN ERROR OFF
ch.qos.logback	TRACE DEBUG INFO WARN ERROR OFF
com	TRACE DEBUG INFO WARN ERROR OFF
com.hazelcast	TRACE DEBUG INFO WARN ERROR OFF
com.hazelcast.core	TRACE DEBUG INFO WARN ERROR OFF
com.hazelcast.core.LifecycleService	TRACE DEBUG INFO WARN ERROR OFF

We'll take a look at the API module in the next section.

## API

This module provides Swagger API documentation for the application's REST API. It also provides a **Try it out** editor for endpoints:

The screenshot shows the JHipster API documentation interface. At the top, there is a navigation bar with links for Home, Entities, Administration, Language, and Account. A red diagonal banner on the left says "Development". The main title is "swagger" and the dropdown shows "default (v2/api-docs)". Below this, the title "store API" is displayed, followed by a subtitle "store API documentation". The "account-resource : Account Resource" section lists the following operations:

Method	Path	Description
GET	/api/account	getAccount
POST	/api/account	saveAccount
POST	/api/account/change-password	changePassword
POST	/api/account/reset-password/finish	finishPasswordReset
POST	/api/account/reset-password/init	requestPasswordReset
GET	/api/activate	activateAccount
GET	/api/authenticate	isAuthenticated
POST	/api/register	registerAccount

Below this, there are sections for "user-jwt-controller : User JWT Controller" and "user-resource : User Resource", each with "Show/Hide", "List Operations", and "Expand Operations" buttons. At the bottom, it says "[ BASE URL: / , API VERSION: 0.0.1 ]".

In the next section, we'll run some tests on the application we just created.

## Running generated tests

Good software development is never complete without good testing. JHipster generates quite a lot of automated tests out of the box, and there are options available so that we can choose even more. Let's run generated server-side and client-side tests for the application to make sure everything is working as expected.

First, open a Terminal/command line and navigate to the project folder.

We'll take a look at server-side tests in the next section.

## Server-side tests

Server-side integration tests and unit tests are present in the `src/test/java` folder.

These can be run directly from the IDE by choosing a package or individual test and running it, or via the command line by running the Gradle `test` and `integrationTest` tasks. Let's run it using the command line. In a new Terminal, navigate to the application source folder and execute the following command. It should finish with a success message, as follows:

```
> ./gradlew test integrationTest
...
BUILD SUCCESSFUL in 36s
10 actionable tasks: 3 executed, 7 up-to-date
```

We'll take a look at client-side tests in the next section.

## Client-side tests

Client-side unit tests and end-to-end tests are available under `src/test/javascript`.

These tests can be run using the available npm scripts or Gradle tasks.



You can see all available Gradle tasks by running `./gradlew tasks`.

Let's run tests using npm scripts. First, let's run Jest unit tests. In the Terminal, execute the following command. You can also use `yarn` instead of `npm` if you choose that during generation:

```
> npm test
```

This should produce a detailed coverage report, followed by output similar to the following:

```
Test Suites: 24 passed, 24 total
Tests: 95 passed, 95 total
Snapshots: 0 total
Time: 19.303s
Ran all test suites.
```

Now, let's run Protractor end-to-end tests using the npm script. To run e2e tests, we need to make sure that the server is running. If you have shut down the server that we started earlier, make sure that you start it again by running `./gradlew` in a Terminal. Now, open a new Terminal, navigate to the application folder, and execute the following command:

```
> npm run e2e
```

This will start the Protractor tests, which will open a new Chrome browser instance and execute tests there. When they've finished, you should see something similar to the following in the console:

```
11 passing (19s)

[00:32:53] I/launcher - 0 instance(s) of WebDriver still running
[00:32:53] I/launcher - chrome #01 passed
```

## Summary

In this chapter, we learned how to create a monolithic web application using JHipster. We also walked through important aspects of the source code we created and learned how to run our application and automated tests. We also browsed the modules that were created and saw them in action.

In the next chapter, we will learn how to utilize JHipster so that it models our business use case and generates entities for it. We will also learn about the **JHipster Domain Language (JDL)**.

# 4

# Entity Modeling with JHipster Domain Language

In the previous chapter, we saw how we can use JHipster to generate a production-grade web application with a lot of awesome features, such as i18n, administration modules, account management, and so on.

In this chapter, we will see how we can enrich that application with business entities and a model. With this, you will be able to design and generate JPA-style entities for your use case, and once you have completed this chapter, you will have added an e-commerce domain model to the application we created earlier, making it a more realistic CRUD application.

We will learn about the following in this chapter:

- JHipster Domain Language (JDL)
- JDL-Studio
- Entity and relationship modeling with JDL
- Entity generation

## Introduction to JDL

JDL (<http://www.jhipster.tech/jdl/>) is used to create the domain model for a JHipster application. It provides a simple and user-friendly **domain-specific language (DSL)** to describe the entities and their relationships.

JDL is the recommended way to create entities for an application and can replace the entity generator provided by JHipster, which can be difficult to use when creating a lot of entities. The JDL is normally written in one or more files with a `.jdl` extension.

Visit <https://www.jhipster.tech/jdl/getting-started> for the complete documentation on JDL.



If you prefer to work with UML and UML modeling tools, then check out JHipster-UML (<http://www.jhipster.tech/jhipster-uml/>), a tool that can create entities from popular UML tools.

Let's learn about the DSL grammar in the next section.

## DSL grammar for JDL

Now, let's look at the JDL grammar. At the time of writing, JDL supports application development, deployment configurations, and complete entity models with relationships and options such as **data transfer objects (DTO)**, service layers, and so on. The grammar can be broken down into the following:

- Application declaration
- Entity declaration
- Relationship declaration
- Option declaration
- Deployment declaration

In the following syntax, [ ] indicates that something is optional and \* indicates that more than one can be specified.

JavaDocs can be added to entity declarations and `/** */` Java comments can be added to fields and relationship declarations. JDL-only comments can be added using the `//` syntax.

It is also possible to define numerical constants in JDL—for example,

`DEFAULT_MIN_LENGTH = 1.`

We will learn about application declaration using JDL later on. For now, let's learn how to declare entities.

## Entity modeling with JDL

Entity declaration is done using the following syntax:

```
[<entity javadoc>]
[<entity annotation>*]
entity <entity name> [<table name>] {
    [<field javadoc>]
    [<field annotation>*]
    <field name> <field type> [<validation>*]
}
```

The `<entity name>` phrase is the name of the entity and will be used for class names and table names. Table names can be overridden using the optional `<table name>` parameter.

The `<field name>` phrase is the name of the fields (attributes) you want for the entity and `<field type>` is the field type, such as `string`, `integer`, and so on. Refer to <https://www.jhipster.tech/jdl/entities-fields#field-types-and-validations> for all supported field types. The ID field will be automatically created and so does not need to be specified in JDL.

The `<validation>` phrase is optional and one or more `<validation>` phrases for the fields can be specified depending on the validation supported by the field type. For validations such as max length and pattern, values can be specified in braces.

The `<entity javadoc>` and `<field javadoc>` phrases are places where you can add documentation using `/** */` syntax to be added to generated Java domain classes.

Multiple annotations can be added to the entity or field using `<entity annotation>` and `<field annotation>`. Refer to <https://www.jhipster.tech/jdl/options> for all available annotations.

An example entity declaration would look like the following:

```
/*
 * This is customer entity Javadoc comment
 * @author Foo
 */
@dto(mapstruct)
entity Customer {
    /** Name field */
    name String required,
    age Integer,
    address String maxlength(100) pattern(/[^a-zA-Z0-9]+/)}
}
```

Enumerations can also be declared using the following syntax:

```
enum [<enum name>] {  
    <ENUM KEY> ([<enum value>])  
}
```

Here is an example:

```
enum Language {  
    ENGLISH, DUTCH, FRENCH  
}
```

The `<enum value>` phrase is optional.

Let's learn about relationship management in the next section.

## Relationship management

The relationship between entities can be declared using the following syntax:

```
relationship <type> {  
    <from entity>[<relationship name>[(<display field>)] <validation>*]<br/>  
    to  
    <to entity>[<relationship name>[(<display field>)] <validation>*]  
}
```

The `<type>` can be `OneToMany`, `ManyToOne`, `OneToOne`, or `ManyToMany`, and as the name suggests, declares the relationship type between `<from entity>` and `<to entity>`.

The `<from entity>` phrase is the name of the owner entity of the relationship or the source. The `<to entity>` phrase is the destination of the relationship.

The `<relationship name>` phrase is optional and can be used to specify the field names to be created for the relationship in the domain object. The `<display field>` phrase can be specified in braces to control the field of the entity to be shown in the drop-down menu on the generated web page; the ID field will be used by default. The `<validation>` phrase can be specified on the `<from entity>` or `<to entity>` and is optional. Currently, the only supported validation is required.

Relationships that are labelled as `OneToMany` and `ManyToMany` are always bidirectional in JHipster. In the case of `ManyToOne` and `OneToOne` relationships, it is possible to create both bidirectional and unidirectional relationships. For unidirectional relationships, just don't enter a `<relationship name>` on the destination/to entity.

Multiple relationships of the same type can be declared within the same block, separated by a comma.

An example relationship declaration would look like the following:

```
entity Book
entity Author
entity Tag

relationship OneToMany {
    Author{book} to Book{writer(name)} required,
    Book{tag} to Tag
}
```

The user is an existing entity in JHipster, and it is possible to have certain relationships with the user. Many-to-many and one-to-one relations can be declared, but the other entity must be the source or owner. Many-to-one relationships are also possible with a user entity.

Let's learn about **Data Transfer Objects (DTO)**, service, and pagination options in the next section.

## DTO, service, and pagination options

JDL also allows us to declare entity-related options easily. Refer to <https://www.jhipster.tech/jdl/options> for more details. The options that are currently supported are as follows:

- **service**: By default, JHipster generates REST resource classes that call the entity repositories directly. This is the simplest option, but in real-world scenarios, we might need a service layer to handle business logic. This option lets us create a service layer with a simple Spring service bean class or with a traditional interface and implementation for the service bean. Possible values are `serviceClass` and `serviceImpl`. Choosing the latter will create an interface and implementation, which is preferred by some people.
- **dto**: By default, domain objects are directly used in the REST endpoints that are created, which may not be desirable in some situations, and you might want to use an intermediary DTO to have more control. JHipster lets us generate the DTO layer using Mapstruct (<http://mapstruct.org/>), an annotation preprocessor library that automatically generates the DTO classes. It is advisable for you to use a service layer when using DTO. A possible value is `mapstruct`. For more information, visit <http://www.jhipster.tech/using-dtos/>.

- `filter`: This option lets us enable JPA-based filtering capabilities for the entity. This works only when a service layer is used. For more details, visit <http://www.jhipster.tech/entities-filtering/>.
- `paginate`: This option lets us enable pagination for an entity. This enables pagination on the resource layer and also implements a paging option on the client-side. Possible values are `no`, `pagination`, and `infinite-scroll`.
- `noFluentMethod`: This lets us disable Fluent-API-styled setters for the generated entity domain objects.
- `skipClient`/`skipServer`: These options let us either skip the client-side or server-side code during generation.
- `angularSuffix`: This option lets us specify a suffix for the folder and class names in the frontend code.
- `readOnly`: This creates a read-only entity.

The general syntax for option declaration is `<OPTION> <ENTITIES | * | all> [with <VALUE>] [except <ENTITIES>]`. All of the options can also be used as annotations on the entities.

The following code shows some possible options and the different syntaxes in which they can be declared:

```
@service(serviceImpl)
entity A
entity B
...
entity Z

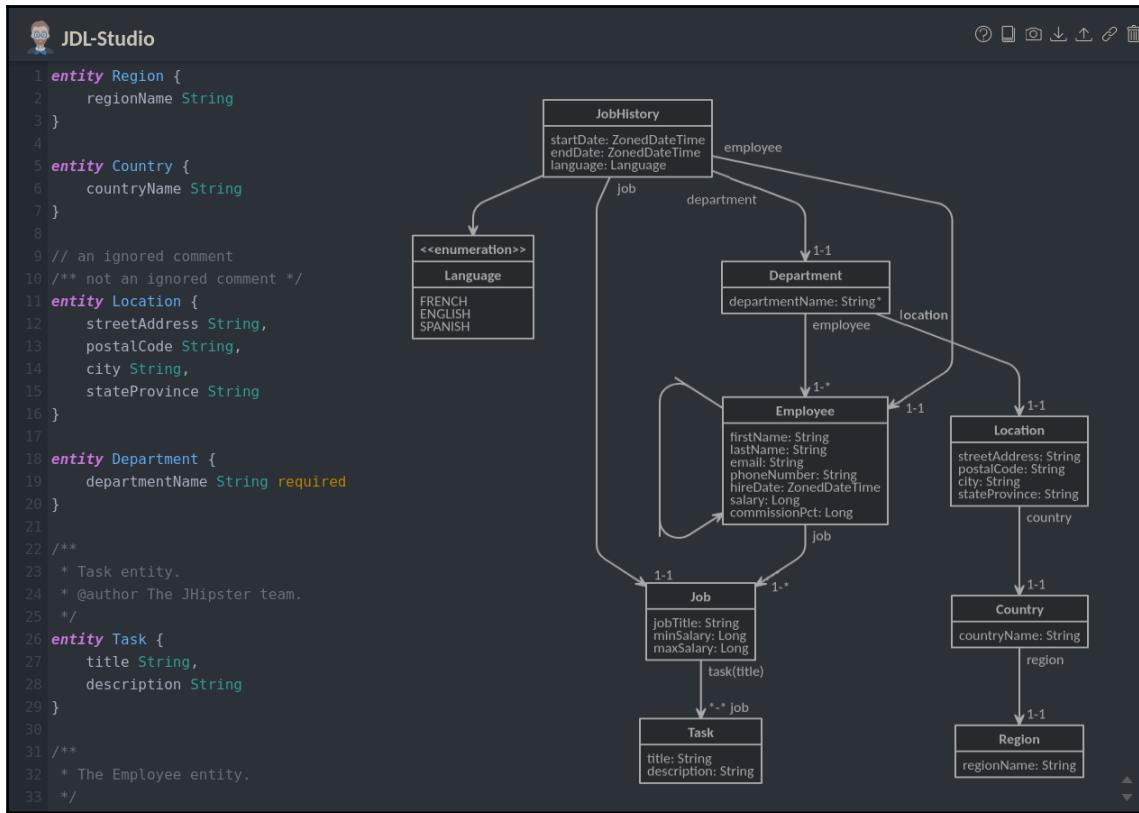
dto * with mapstruct
service B with serviceClass
paginate * with pagination except B, C
paginate B, C with infinite-scroll
filter A, B
```

Let's take a look at the JDL-Studio in the next section.

## JDL-Studio

We will be using JDL-Studio (<https://start.jhipster.tech/jdl-studio/>) to create our JDL file. It is an online web application built by the JHipster team for creating JDL files in a visual editor. The tool shows a visual representation of the created entity model and also lets you import/export JDL and capture image snapshots.

You can create an account to save your models under, as well:



The tool also provides features such as syntax highlighting, autocompletion, error reporting, and Sublime Text-style keyboard shortcuts.

Go to <https://start.jhipster.tech/jdl-studio/> to open the application.



Please note that by default, this application stores the JDL in your browser's local storage. You can create an account with JHipster online if you want to save your JDL files to the cloud.

Let's take a look at our use case and the entity model in the next section.

## Use case entity model

Now, let's look at our use case and the entity model. Before that, clear the default JDL in the JDL-Studio editor.

## Entities

Let's start by defining our entities:

1. Copy the following snippet for `Product` and `ProductCategory` into the JDL-Studio editor:

```
/** Product sold by the Online store */
entity Product {
    name String required
    description String
    price BigDecimal required min(0)
    size Size required
    image ImageBlob
}

enum Size {
    S, M, L, XL, XXL
}

entity ProductCategory {
    name String required
    description String
}
```

The `Product` entity is the core of the domain model; it holds product information such as the `name`, `description`, `price`, `size`, and `image`, which is a blob.

The `name`, `price`, and `size` are required fields. The `price` also has a minimum value validation. The `size` field is an `enum` with defined values.

The `ProductCategory` entity is used to group products together. It has the `name` and `description` fields, where `name` is a required field.

2. Add the following snippet for `Customer` into the JDL-Studio editor:

```
entity Customer {  
    firstName String required  
    lastName String required  
    gender Gender required  
    email String required pattern(/^[^@\s]+@[^\s]+\.\[^@\s]+$/)  
    phone String required  
    addressLine1 String required  
    addressLine2 String  
    city String required  
    country String required  
}  
  
enum Gender {  
    MALE, FEMALE, OTHER  
}
```

The `Customer` entity holds details of the customers using the online shopping portal. Most of the fields are marked as required, and the `email` field has regex pattern validation. The `gender` field is an enum. This entity is related to the system user, which we will see in more detail soon.

3. Add the following snippet for `ProductOrder` and `OrderItem` into the JDL-Studio editor:

```
entity ProductOrder {  
    placedDate Instant required  
    status OrderStatus required  
    code String required  
}  
  
enum OrderStatus {  
    COMPLETED, PENDING, CANCELLED  
}  
  
entity OrderItem {  
    quantity Integer required min(0)  
    totalPrice BigDecimal required min(0)  
    status OrderItemStatus required  
}  
  
enum OrderItemStatus {  
    AVAILABLE, OUT_OF_STOCK, BACK_ORDER  
}
```

The `ProductOrder` and `OrderItem` entities are used to track product orders made by customers. The `ProductOrder` entity holds the `placedDate`, `status`, and `code` of the order, which are all required fields, while `OrderItem` holds information about the `quantity`, `totalPrice`, and `status` of individual items. All fields are required and the `quantity` and `totalPrice` fields have a minimum value validation. The `OrderStatus` and `OrderItemStatus` fields are enum fields.

4. Add the following snippet for `Invoice` and `Shipment` into the JDL-Studio editor:

```
entity Invoice {
    date Instant required
    details String
    status InvoiceStatus required
    paymentMethod PaymentMethod required
    paymentDate Instant required
    paymentAmount BigDecimal required
}

enum InvoiceStatus {
    PAID, ISSUED, CANCELLED
}

enum PaymentMethod {
    CREDIT_CARD, CASH_ON_DELIVERY, PAYPAL
}

entity Shipment {
    trackingCode String
    date Instant required
    details String
}
```

The `Invoice` and `Shipment` entities are used to track the invoice and shipping for the product orders, respectively. Most of the fields in `Invoice` are required and the `status` and `paymentMethod` fields are enums.

The enumerations are used to contain the scope of certain fields, which gives more granular control over those fields.

## Relationships

Now that we have defined our entities, let's add relationships between them:

1. Add the following snippet for relationships into the JDL-Studio editor:

```
relationship OneToOne {  
    Customer{user} to User  
}
```

The first relationship declared is a unidirectional `OneToOne` between a `Customer` entity and the inbuilt `User` entity:

```
Customer (1) -----> (1) User
```

It means that the `Customer` entity knows about the `User` and is the owner of the relationship, but the `User` doesn't know about the `Customer`, and so we will not be able to obtain customers from a `User`. This lets us map customers to the `User` entity and use it for authorization purposes later, ensuring that one customer can be mapped to only one system user.

2. Add the following snippet for relationships into the JDL-Studio editor:

```
relationship ManyToOne {  
    OrderItem{product} to Product  
}
```

This one declares a unidirectional `ManyToOne` relationship from `OrderItem` to `Product`:

```
OrderItem (*) -----> (1) Product
```

This means that the `OrderItem` knows their `Product`, but the `Product` does not know about the `OrderItem`. This keeps the design clean as we don't want to know about orders from products for this use case. In the future, if we want to know the orders that have been made for a product, then we could make this bidirectional.

3. Add the following snippet for a relationship into the JDL-Studio editor:

```
relationship OneToMany {
    Customer{order} to ProductOrder{customer},
    ProductOrder{orderItem} to OrderItem{order},
    ProductOrder{invoice} to Invoice{order},
    Invoice{shipment} to Shipment{invoice},
    ProductCategory{product} to Product{productCategory}
}
```

This declaration is interesting, as we have multiple OneToMany declarations:

```
Customer (1) <-----> (*) ProductOrder
ProductOrder (1) <-----> (*) OrderItem
ProductOrder (1) <-----> (*) Invoice
Invoice (1) <-----> (*) Shipment
ProductCategory (1) <-----> (*) Product
```

They are all bidirectional, meaning that both the source entity and the destination entity know about each other.

We declare that a Customer can have multiple instances of ProductOrder, ProductOrder can have multiple OrderItem and invoices, Invoice can have many instances of Shipment, and ProductCategory can have many products. From the destination entity, the source entities are mapped as ManyToOne.

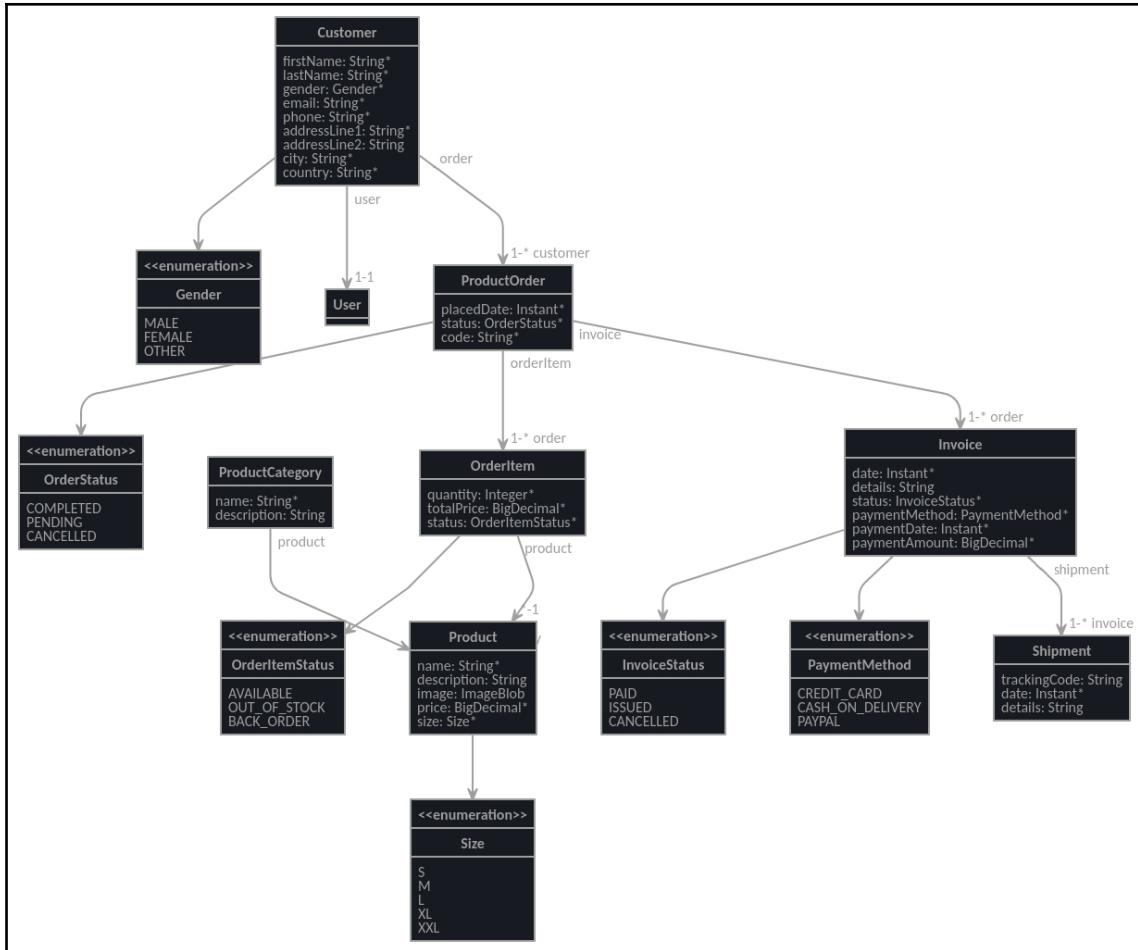
## Options for entities

Add the following snippet for options into the JDL-Studio editor:

```
service * with serviceClass
paginate Product, Customer, ProductOrder, Invoice, Shipment, OrderItem with
pagination
```

In the options, we keep it simple and declare that we want a service class for all entities. We also enable pagination for some of the entities that may get a lot of entries over time.

The following diagram shows the complete model, with all the entities and their relationships as shown in JDL-Studio:



Now, let's download this JDL file to our file system:

1. Click on the **Download** button in the upper-right corner of the JDL-Studio application.
2. Save the file with the name `online-store.jdl` inside the `online-store` directory where we created our application in the previous chapter.

In the next section, we'll learn about entity generation with JHipster.

# Entity generation with JHipster

Now it's time to generate the domain model with our JDL. We will use the `import-jdl` command from JHipster for this.

Open your favorite Terminal application and navigate to the `online-store` folder where we created the application earlier. Then, execute the `import-jdl` command:

```
> cd online-store  
> jhipster import-jdl online-store.jdl
```

This will trigger the entity creation process, and you will be asked to confirm that the system should overwrite existing files with changes. Take a look at the following screenshot:

```
Found the .jhipster/OrderItem.json configuration file, entity can be automatically generated!  
  
The entity OrderItem is being updated.  
  
Found the .jhipster/Invoice.json configuration file, entity can be automatically generated!  
  
The entity Invoice is being updated.  
  
Found the .jhipster/Shipment.json configuration file, entity can be automatically generated!  
  
The entity Shipment is being updated.  
  
create src/main/resources/config/liquibase/changelog/20190818145106_added_entity_Product.xml  
create src/main/resources/config/liquibase/fake-data/product.csv  
create src/main/resources/config/liquibase/changelog/20190818145106_added_entity_constraints_Product.xml  
create src/main/resources/config/liquibase/fake-data/blob/hipster.png  
create src/main/java/com/mycompany/store/domain/Product.java  
create src/main/java/com/mycompany/store/repository/ProductRepository.java  
create src/main/java/com/mycompany/store/web/rest/ProductResource.java  
create src/main/java/com/mycompany/store/service/ProductService.java  
create src/test/java/com/mycompany/store/web/rest/ProductResourceIT.java  
conflict src/main/resources/config/liquibase/master.xml  
? Overwrite src/main/resources/config/liquibase/master.xml? (ynaxdH) _
```

Enter `a` to confirm the overwriting of all files with changes. Once the files are generated, JHipster will trigger an `npm run webpack:build` step to rebuild the client-side code. Once this is done, you will see a success message like the following:

```
DONE Compiled successfully in 10429ms

752 modules
INFO! Congratulations, JHipster execution is complete!
```

Running `git status` on the Terminal shows us that five files were modified and a lot of new files were added. Let's commit the changes to Git. Execute the commands shown in the following code:

```
> git add --all
> git commit -am "generated online store entity model"
```

In the next section, we'll take a look at the code that was generated.

## Generated code walkthrough

Now let's take a look at what has been generated. Let's open the application code in our favorite IDE/editor and look at what has been generated for the `Product` entity.

You might have noticed that there is a `.jhipster` folder at the root of the project, and if you look inside it, you will see a bunch of JSON files. Let's look at `Product.json`; it holds metadata about the generated entity and is used by JHipster to regenerate and edit an entity when needed:

```
{
  "name": "Product",
  "fields": [
    {
      "fieldName": "name",
      "fieldType": "String",
      "fieldValidateRules": [
        "required"
      ]
    },
    {
      ...
    },
    ...
  ],
  "relations": [
    {
      "name": "category"
    }
  ],
  "events": [
    {
      "name": "productCreated"
    }
  ],
  "audited": true,
  "auditedBy": "AuditEvent"
}
```

```
{  
    "fieldName": "price",  
    "fieldType": "BigDecimal",  
    "fieldValidateRules": [  
        "required",  
        "min"  
    ],  
    "fieldValidateRulesMin": 0  
,  
{  
    "fieldName": "size",  
    "fieldType": "Size",  
    "fieldValues": "S,M,L,XL,XXL",  
    "fieldValidateRules": [  
        "required"  
    ]  
,  
{  
    "fieldName": "image",  
    "fieldType": "byte[]",  
    "fieldTypeBlobContent": "image"  
}  
,  
"relationships": [  
    {  
        "relationshipType": "many-to-one",  
        "otherEntityName": "productCategory",  
        "otherEntityRelationshipName": "product",  
        "relationshipName": "productCategory",  
        "otherEntityField": "id"  
    }  
,  
    "changelogDate": "20191207131119",  
    "javadoc": "Product sold by the Online store",  
    "entityTableName": "product",  
    "dto": "no",  
    "pagination": "pagination",  
    "service": "serviceClass",  
    "jpaMetamodelFiltering": false,  
    "fluentMethods": true,  
    "readOnly": false,  
    "clientRootFolder": "",  
    "applications": "*"  
}
```

## Server-side source code

Now let's look at the server-side code that was generated.

### Domain class for the entity

In the `src/main/java/com/mycompany/store/domain` folder, you will find the entity domain object. Open `Product.java`:

```
@ApiModel(description = "Product sold by the Online store")
@Entity
@Table(name = "product")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    @Column(name = "name", nullable = false)
    private String name;

    @Column(name = "description")
    private String description;

    @NotNull
    @DecimalMin(value = "0")
    @Column(name = "price", precision = 21, scale = 2, nullable =
        false)
    private BigDecimal price;

    @NotNull
    @Enumerated(EnumType.STRING)
    @Column(name = "size", nullable = false)
    private Size size;

    @Lob
    @Column(name = "image")
    private byte[] image;

    @Column(name = "image_content_type")
    private String imageContentType;

    @ManyToOne
```

```
@JsonIgnoreProperties("products")
private ProductCategory productCategory;

// jhipster-needle-entity-add-field - JHipster will add fields
// here, do not remove

... // getters

public Product name(String name) {
    this.name = name;
    return this;
}

... // setters

// jhipster-needle-entity-add-getters-setters - JHipster will add
// getters and setters here, do not remove

... // equals, hashCode and toString methods
}
```

The entity class defines the fields and relationships. The `ApiModel` annotation is used by Swagger to show useful documentation when the entity is used in an endpoint:

```
@ApiModel(description = "Product sold by the Online store")
```

The following code phrases are JPA annotations declaring the **Plain Old Java Object (POJO)** as an entity and mapping it to an SQL table:

```
@Entity
@Table(name = "product")
```

The following is a Hibernate annotation, which lets us enable a level 2 cache for this entity. In our case, we will be using Hazelcast:

```
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
```

The `id` field is special and is mapped as a generated value field. Depending on the **database (DB)**, this field will use a native generation technique or a sequence provided by Hibernate. Since we are using MySQL, it will use the native DB primary key generation technique:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

The following JPA annotation is used to map columns to fields, and it can also be used to declare properties such as nullable, precision, scale, unique, and so on for the field:

```
@Column(name = "name", nullable = false)
```

The following are bean validation annotations that enable validation for the fields:

```
@NotNull  
@DecimalMin(value = "0")
```

The image field is a blob and it is marked by the `Lob` type since we are using MySQL. It also has an additional field to hold the content type information:

```
@Lob  
@Column(name = "image")  
private byte[] image;  
  
@Column(name = "image_content_type")  
private String imageContentType;
```

The `Enumerated annotation` is used to map `enum` fields. These are stored as simple `varchar` fields in the DB:

```
@Enumerated(EnumType.STRING)
```

The relationships are mapped using annotations such as `@ManyToOne`, `@OneToOne`, `@OneToOne`, and `@ManyToMany`:

```
@ManyToOne  
private ProductCategory productCategory;
```

Here, `ProductCategory` is mapped as `ManyToOne`; on the other side of the relationship, `Product` is mapped as `OneToMany`, as shown here:

```
@OneToMany(mappedBy = "productCategory")  
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)  
private Set<Product> products = new HashSet<>();
```

As you can see, the relationship also specifies a cache for it. The following is a fluent setter generated by default, along with the standard setter:

```
public Product name(String name) {  
    this.name = name;  
    return this;  
}
```



Fluent methods can be turned off by specifying `noFluentMethod` for the entity in JDL. Fluent methods are handy as they let us use chain setters for more concise code, such as new

```
Product().name("myProduct").price(10);
```

The corresponding table definitions and constraints are created using Liquibase and can be found in `src/main/resources/config/liquibase/changelog` with the filenames

`<timestamp>_added_entity_Product` and

`<timestamp>_added_entity_constraints_Product.xml`, which automatically get applied to the database when we reload or start the application again.

## Repository interface for the entity

In the `src/main/java/com/mycompany/store/repository` folder, you will find the entity repository service. Open `ProductRepository.java`:

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

The repository service is just an empty interface that extends the `JpaRepository` class. Since it is a Spring Data repository, the implementation is automatically created, allowing us to perform all CRUD actions using this simple interface declaration. Additional repository methods can be added here easily. We will learn more about this in the next chapter.

## Service class for the entity

Since we opted to generate service classes for our entities, let's look at one. In the `src/main/java/com/mycompany/store/service` folder, you will find the entity repository service. Open `ProductService.java`:

```
@Service
@Transactional
public class ProductService {

    private final Logger log =
        LoggerFactory.getLogger(ProductService.class);

    private final ProductRepository productRepository;

    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
```

```
    }  
  
    ...  
}
```

The service uses constructor injection to get its dependencies, which are automatically injected by Spring during bean instantiation. The service is also marked as `@Transactional` to enable transaction management for data access. The service defines CRUD action methods—for example, the `findAll` method calls the equivalent repository method while adding a read-only transaction rule to it. You can see that the method already supports pagination and returns the results as `Page`. The `Page` and `Pageable` objects are provided by Spring and let us easily control pagination:

```
@Transactional(readOnly = true)  
public Page<Product> findAll(Pageable pageable) {  
    log.debug("Request to get all Products");  
    return productRepository.findAll(pageable);  
}
```

## Resource class for the entity

In the `src/main/java/com/mycompany/store/web/rest` folder, you will find the entity resource service. Open `ProductResource.java`:

```
@RestController  
@RequestMapping("/api")  
public class ProductResource {  
    ...  
}
```

The resource acts as the controller layer, and in our case, it serves the REST endpoints to be used by our client-side code. The endpoint has a base mapping to `"/api"`:

```
@GetMapping("/products")  
public ResponseEntity<List<Product>> getAllProducts(Pageable  
pageable) {  
    log.debug("REST request to get a page of Products");  
    Page<Product> page = productService.findAll(pageable);  
    HttpHeaders headers = PaginationUtil.  
        generatePaginationHttpHeaders(  
            ServletUriComponentsBuilder.fromCurrentRequest(),  
            page);  
    return ResponseEntity.ok().headers(headers).  
        body(page.getContent());  
}
```

All the CRUD actions have equivalent mapping methods here—for example, the `getAllProducts` maps to `findAll` from our service. The resource also handles pagination by adding appropriate headers for pagination.

Let's take a look at the client-side code in the next section.

## Client-side

The client-side resources for the entity are created in the `src/main/webapp/app/entities` folder. Let's take a look at the code created for the `Product` entity in the `product` folder.

### TypeScript model class for the entity

Let's look at the TypeScript model generated in `src/main/webapp/app/shared/model/product.model.ts`. This maps directly to the domain object:

```
export interface IProduct {
    id?: number;
    name?: string;
    description?: string;
    price?: number;
    size?: Size;
    imageContentType?: string;
    image?: any;
    productCategory?: IProductCategory;
}

export class Product implements IProduct {
    constructor(
        public id?: number,
        public name?: string,
        public description?: string,
        public price?: number,
        public size?: Size,
        public imageContentType?: string,
        public image?: any,
        public productCategory?: IProductCategory
    ) {}
}
```

The fields are all optional, making it possible to create an object instance without any values. You will also see that the enums are generated alongside the model in the file.

## Angular services for the entity

The `ProductService` is an Angular service that interacts with our REST endpoints, and is created in `product.service.ts`:

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

    private resourceUrl = SERVER_API_URL + 'api/products';

    constructor(private http: HttpClient) { }

    ...

    query(req?: any): Observable<EntityArrayResponseType> {
        const options = createRequestOption(req);
        return this.http.get<IProduct[]>(
            this.resourceUrl,
            { params: options, observe: 'response' }
        );
    }

    ...
}
```

As you can see, the service has a constructor with dependencies that are injected following a similar pattern as our server-side code. There are methods mapping all the CRUD actions to the backend REST resource. The HTTP calls make use of RxJS observables to provide an asynchronous streaming API, which is much better than a promise-based API.

## Angular components of the entity

For an entity, there are four component classes generated in four files and four HTML files that are used in the components.

The `ProductComponent`, defined in `product.component.ts`, handles the main listing screen. It uses `product.component.html` as the template. The component manages the view and their actions. It also calls multiple services to fetch data and perform other actions, such as alerts and event broadcasts:

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

The `ProductDetailComponent` handles the detail view screen using `product-detail.component.html` as the template and is defined in `product-detail.component.ts`.

The `ProductDeleteDialogComponent`, defined in `product-delete-dialog.component.ts`, manages the delete pop-up dialog using `product-delete-dialog.component.html` as the template.

## Angular route for the entity

We need a route declaration so that we can access the entity pages. This is declared in `product.route.ts`.

For example, this declares the detail view of the entity:

```
{
    path: ':id/view',
    component: ProductDetailComponent,
    resolve: {
        product: ProductResolve
    },
    data: {
        authorities: ['ROLE_USER'],
        pageTitle: 'storeApp.product.home.title'
    },
    canActivate: [UserRouteAccessService]
},
```

The data attribute is used to pass metadata, such as allowed roles and page titles, to the component. `UserRouteAccessService`, defined in the `canActivate` attribute, decides whether a user has the authorization to view the page and uses the authorities' metadata and authentication details to verify the user's authorization. `ProductResolve` is used to determine the function to be used for resolving the route.

## Angular module for the entity

Lastly, we have a module for the entity. Angular modules can be used to consolidate all components, directives, pipes, and services of an entity so that they can be imported into other modules easily. The `StoreProductModule` module is defined in `product.module.ts`:

```
@NgModule({
  imports: [StoreSharedModule, RouterModule.forChild(productRoute)],
  declarations: [
    ProductComponent,
    ProductDetailComponent,
    ProductUpdateComponent,
    ProductDeleteDialogComponent
  ],
  entryComponents: [ProductDeleteDialogComponent]
})
export class StoreProductModule {}
```

The module declares the components that are used by it. The module also imports shared modules so that it can access shared services and components. The module is lazily loaded by the `StoreEntityModule` defined in `entity.module.ts` under `src/main/webapp/app/entities`.

In the next section, we'll take a look at the pages that will be generated.

## Generated pages

Let's start the application to view the generated pages. In the Terminal, execute the Gradle command, as follows:

```
> ./gradlew
```

This will start the server in development mode locally. Since the `import-jdl` step already compiled the frontend code, we don't have to run `npm start` just to see the new pages, but please note that for further development, it is better to use `npm start` along with the preceding command. If you had the server already running while generating the entities, then there is no need to run this command; instead, just compile the source again using the `./gradlew compileJava` command or using your IDE and Spring DevTools will hot reload the application for you. If you had `npm start` running, then a hot reload will take place on the client-side as well; otherwise, just refresh the page. We will learn more about hot reloading in the next chapter.

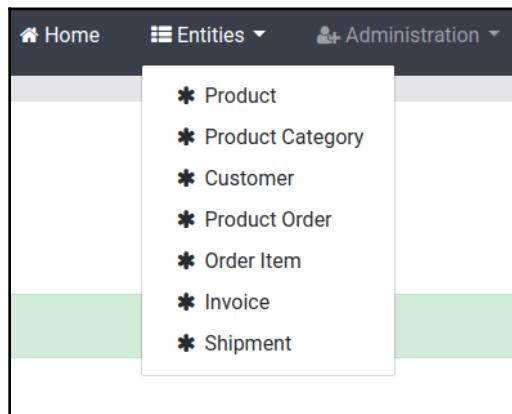
Once you see the following message, the server is ready and we can navigate to the URL `http://localhost:8080` in our favorite browser:

---

```
Application 'store' is running! Access URLs:
Local: http://localhost:8080
External: http://192.168.2.7:8080
Profile(s): [swagger, dev]
```

---

If you are not already logged in, then do so using the default admin user with the password `admin` by clicking on the **Sign in** link on the home page. Once logged in, click on the **Entities** link in the menu and you will see all our entities listed there:



Click on the **Product** and you will see the **Products** listing screen. It has some fake data autogenerated for development purposes, which is loaded only for the **dev** profile:



ID	Name	Description	Price	Size	Image	Product Category
1	mobile Fish	Home Loan Account Table Computer	5808	XL	 image/png, 27 702 bytes	<button>View</button> <button>Edit</button> <button>Delete</button>
2	deposit Tasty Metal Bacon Implementation	Chips	18340	XL	 image/png, 27 702 bytes	<button>View</button> <button>Edit</button> <button>Delete</button>
3	Home Namibia	Steel zero administration	36636	L	 image/png, 27 702 bytes	<button>View</button> <button>Edit</button> <button>Delete</button>
4	Savings Account syndicate Planner	Refined Rubber Chicken Table	6505	XL	 image/png, 27 702 bytes	<button>View</button> <button>Edit</button> <button>Delete</button>

The loading of the fake data can be disabled by removing the **faker** profile in the `spring.liquibase.contexts` property in the application configuration file at `src/main/resources/config/application-dev.yml`. You can also disable the loading of fake data during the initial app creation by passing the `--skip-fake-data` flag to the `jhipster` command.

Let's create an entity. Click on the **Create a new Product** button on the screen and you will see the following **Create or edit a Product** page:

### Create or edit a Product

Name

Description

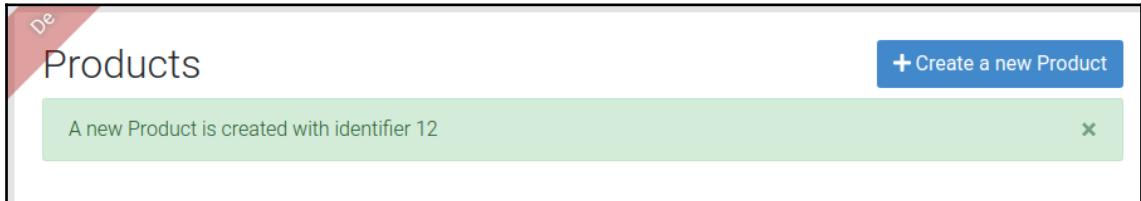
Price

Size

Image  
  
image/jpeg, 661 426 bytes [X]  
 tshirt\_bleu\_jhipster17.jpg

Product Category

Enter the **Name**, **Description**, **Price**, and **Size**. Select an image by clicking on the **Choose** file button and choosing a **Product Category**. Now click on **Save** and the listing screen will be refreshed with the following success message:



The **Products** screen now shows our new entity with buttons for **View**, **Edit**, and **Delete**. There are also pagination links on the bottom. Explore the **View**, **Edit**, and **Delete** buttons by clicking on each of them.

In the next section, we'll run the test code that we generated.

## Running the generated tests

Let's run all the tests to make sure that the generated test code works fine.

Let's run the server-side unit/integration tests, client-side Jest unit tests, and Protractor e2e tests using the command line. In a new Terminal, navigate to the application source folder and execute these commands. They should finish with a success message:

```
> ./gradlew test integrationTest && npm test && npm run e2e
```

Make sure you have the application running, as e2e tests will need it. If the application is not running, then first start it by running `./gradlew` in a Terminal.

## Summary

In this chapter, we learned how to model and create entities using JDL. We also walked through some important features of the created source code and then browsed through the created entity modules and saw them in action.

In the next chapter, we will learn how we can utilize JHipster to further develop the application and include specific business logic and tweaks. We will also look more deeply into some of the technologies that we have used.

# 5

# Customization and Further Development

In the previous chapter, we saw how to use the **JHipster Domain Language (JDL)** to model and generate our domain model. We also learned about entity relationships and the `import-jdl` sub-generator. In this chapter, we will see how we can further customize and add business logic to the generated online-store application to suit our needs. We will learn about the following:

- Live reload with Spring DevTools and BrowserSync
- Customizing the Angular frontend for an entity
- Editing an entity that was created using the JHipster entity generator
- Changing the look and feel of the application using a Bootstrap theme
- Adding a new i18n language using the JHipster language generator
- Customizing the generated REST API to add additional role-based authorization with Spring Security
- Creating new Spring Data JPA queries and methods

## Live reload for development

One of the most annoying and time-consuming aspects of developing an application is recompiling the code and restarting the servers to see the code changes that we have made. Traditionally, JavaScript code used to be easier, as it didn't need any compilation and you could just refresh the browser and see the changes; however, even though current MVVM stacks make the client-side more important than before, they also introduce side effects, such as the transpiling of client-side code, and more. So, if you were refactoring a field for an entity, you would traditionally need to perform the following tasks to see the changes in your browser:

1. Compile the server-side Java code.
2. Apply the table changes to the database.
3. Recompile the client-side code.
4. Restart the application server.
5. Refresh the browser.

This takes a lot of time, is frustrating to do for every small change, and results in you making more changes before checking the previous change, thereby affecting productivity.

But what if I told you that you don't have to do any of these and that all of this could happen automatically as you save your changes while using your IDE? That would be awesome, wouldn't it?

With JHipster, you get exactly that. JHipster uses Spring Boot DevTools, the webpack dev server, and BrowserSync to enable a nice, live reload feature for the end-to-end code.

Let's take a quick look at the technologies used.

## Spring Boot DevTools

Spring Boot DevTools (<https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-devtools.html>) enables Spring Boot applications to reload the embedded server when there is a change in the classpath. It states the following:

*The aim of this module is to try and improve the development-time experience when working on Spring Boot applications.*

And it does exactly that. It uses a custom classloader to restart the application when a class is updated and recompiled, and since the server is hot reloaded, it is much faster than a cold restart.

It isn't as cool as JRebel or similar technologies, which perform instant reloads, but it beats doing it manually, and doesn't require any extra configuration to work.

JHipster automatically enables DevTools in the `dev` profile and using an IDE that can automatically recompile classes on save, DevTools will ensure that the application is reloaded and is up to date. Since Liquibase is used, any schema updates using proper changelogs will also get updated. Make sure that you do not change existing changelogs, as this will cause a checksum error. Application reloads can also be triggered by simply using the `mvnw compile` or `gradlew compileJava` commands depending on the build tool used.



If you choose a NoSQL DB, such as MongoDB, Cassandra, or Couchbase, then bear in mind that JHipster provides database migration tools for these as well.

## Webpack dev server and BrowserSync

Webpack dev server (<https://github.com/webpack/webpack-dev-server>) provides a simple express server using webpack dev middleware and supports live reloads when assets change. Webpack dev middleware supports features such as hot module replacement and in-memory file access.



In Webpack version 4 and above, a new alternative called `webpack-serve` (<https://github.com/webpack-contrib/webpack-serve>) is used instead of Webpack dev server. It uses native WebSocket support in newer browsers.

BrowserSync (<https://browsersync.io/>) is a Node.js tool that helps with browser testing by synchronizing the file changes and interactions of the web page across multiple browsers and devices. It provides features such as auto-reloads on file changes, synchronized UI interactions, scrolling, and so on. JHipster integrates BrowserSync with Webpack dev server to provide a productive development setup. It makes testing a web page on different browsers and devices super easy. Changes to CSS are loaded without a browser refresh.

To use live reloads on the client-side, you need to run `npm start`, which will start the development server and open up a browser pointing to `http://localhost:9000`. Note port `9000` in the URL. BrowserSync will be using this port, while the application backend will be served at `8080`, and all requests will be proxied through webpack dev middleware.



Open another browser—for example, Firefox—if BrowserSync has opened Chrome already or vice versa. Now place them side by side and play around with the application. You will see that your actions are replicated, thanks to BrowserSync. Try changing some code and save the file to see the live reload in action. Note that you need to be authenticated in both of the browsers, and if you perform a `POST` request, it will be done twice as there are two browsers performing the action.

## Setting up live reloads for an application

Let's start the perfect development setup for the application we created. In a Terminal, start the server in dev mode by running `./gradlew`, and in another Terminal, start the client-side development server by running `npm start`.

Now, when you make any changes on the server-side, simply run `./gradlew compileJava`, or if you are using an IDE, click on the **Compile** button.



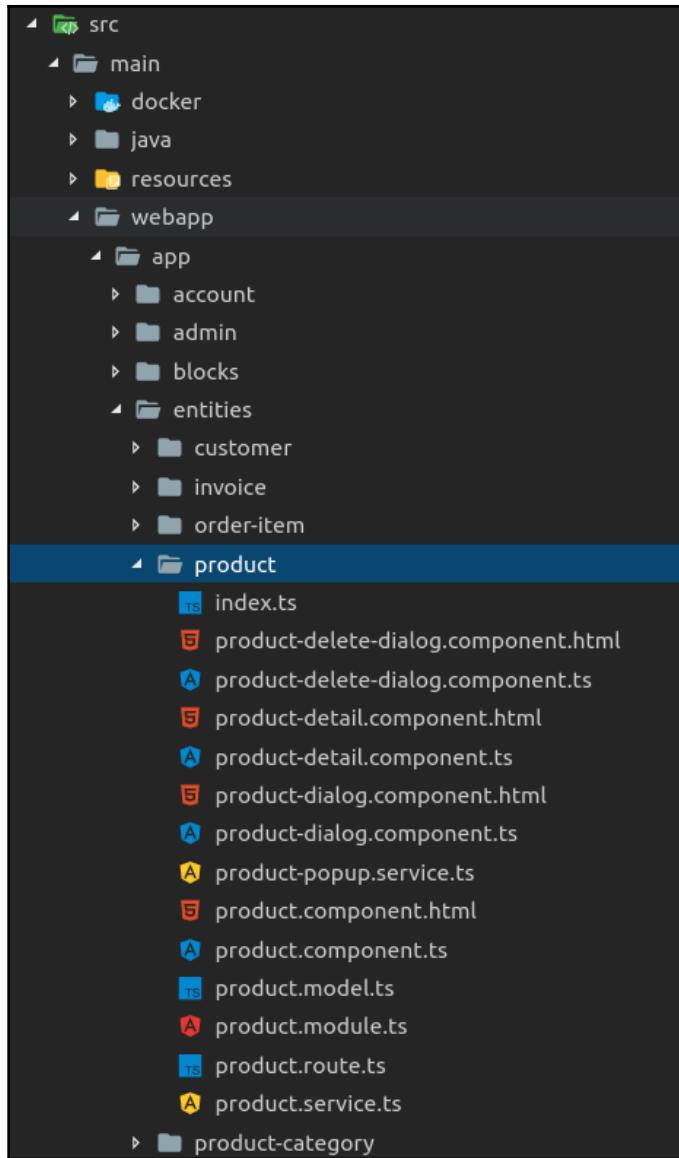
With IntelliJ IDEA, files are automatically saved, and so you can press `Ctrl + S` to compile the classes, giving you a nice workflow. In Eclipse, saving a class automatically compiles it.

When you make changes on the client-side, simply save the file and webpack dev server and BrowserSync will do the rest. Now let's see how we can customize an entity.

## Customizing the Angular frontend for an entity

Now that we have our entity domain model created and working, let's make it more usable. The **product** listing screen has a table view generated by JHipster; it is sufficient for simple CRUD operations, but isn't the best-suited user experience for end users who want to browse our product listing. Let's see how we can easily change this to something more appealing. We will also add a nice client-side filter option to filter the listing. We will be using both Angular and Bootstrap features for this:

1. First, let's find the source code that we would need to edit. In your favorite editor/IDE, navigate to `src/main/webapp/app/entities/product`:



Let's start by customizing the `product.component.html` file to update the UI view of the product listing.

The HTML code currently renders a table view and uses some Angular directives to enhance the view with sorting and pagination. Let's change the view from a table into a list, but first, if it's not already open, then open the development web server from BrowserSync by navigating to `http://localhost:9000`.

2. If you disabled fake data during code generation, then log in and navigate to **Entities | Product Category**, and create a category. Then navigate to **Entities | Product** and create a few new products so that we have something to list:

Products							<a href="#">+ Create a new Product</a>
ID	Name	Description	Price	Size	Image	Product Category	
1	JHipster T-Shirt-Men	JHipster T-Shirt for men	20	M		1	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
2	JHipster T-shirt-Women	JHipster T-shirt for Women	20	S		1	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
3	Spring T-shirt	Spring T-shirt womens	15	M		1	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
4	Spring T-shirt-Men	Spring T-shirt for men	10	XL		1	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>

Showing 1 - 4 of 4 items.  
[««](#) [«](#) [1](#) [»](#) [»»](#)

3. We can use the Bootstrap list group (<https://getbootstrap.com/docs/4.0/components/list-group/>) component for this purpose.

Let's use the following snippet and change the view. Replace the `div` tag with `class="table-responsive"` with the following code:

```
<div *ngIf="products">
  <div class="list-group">
    <a class="list-group-item list-group-item-action flex-column align-items-start"
      *ngFor="let product of products; trackBy: trackId">
      <div class="d-flex w-100 justify-content-between">
        <a [routerLink]=["/product", product.id, 'view']>
          <h5 class="mb-1">{{ product.name }}</h5>
        </a>
        <small *ngIf="product.productCategory">
          <a [routerLink]=["/product-category", product
            .productCategory?.id, 'view']>
            {{ product.productCategory?.id }}
          </a>
        </small>
      </div>
    </a>
  </div>
```

```
<small class="mb-1">{{ product.description }}</small>
<p class="mb-1">Price: {{ product.price }}</p>
<small>
  Size:
  <span jhiTranslate="{{ 'storeApp.Size.' + product.size
}}">
    {{ product.size }}
  </span>
</small>
</a>
</div>
</div>
```

As you can see, we are iterating the products using the Angular directive, `*ngFor="let product of products; trackBy: trackId"`, on the anchor element so that the element is created for each product in the list. We wrap this in a `*ngIf="products"` directive so that the view is rendered only when the product's object is defined. The `[routerLink]="/product', product.id, 'view'"` directive will create a `href` tag for the anchor using the Angular router so that we can navigate to the particular product route. We then use properties from the product in template strings to be rendered using the `{{product.name}}` syntax. As you save the code, you might notice that the view refreshes automatically, thanks to BrowserSync.



The `trackBy` function used in `ngFor` lets Angular decide which items are added or removed from a collection. This improves rendering performance as Angular can now figure out which items need to be added or removed from the **Document Object Model (DOM)** exactly, without having to recreate the entire collection. Here, we provide `trackId` as the function to uniquely identify an item in the collection.

This will produce the following:

The screenshot shows a web application interface for managing products. At the top, there's a header with the word "Products" and a blue button labeled "+ Create a new Product". Below the header, there are four product entries, each with a small orange number "1" in the top right corner:

- JHipster T-Shirt-Men**  
JHipster T-Shirt for men  
**Price: 20**  
Size: M
- JHipster T-shirt-Women**  
JHipster T-shirt for Women  
**Price: 20**  
Size: S
- Spring T-shirt**  
Spring T-shirt women's  
**Price: 15**  
Size: M
- Spring T-shirt-Men**  
Spring T-shirt for men  
**Price: 10**  
Size: XL

At the bottom of the list, it says "Showing 1 - 4 of 4 items." and features a pagination control with buttons for "««", "«", "1", "»", and "»»".

While it's a good start, it's not enough. So, let's go in and make it better.

4. Let's add the image to the listing first. Modify the code to add Bootstrap rows and columns, as shown in the following code. The original code that renders the content is moved into the second column and remains unchanged:

```
<div *ngIf="products">
  <div class="list-group">
    <a class="list-group-item list-group-item-action flex-column align-items-start"
       *ngFor="let product of products; trackBy: trackId">
      <div class="row">
        <div class="col-2 col-xs-12 justify-content-center">
          <img [src]="'data:' + product.imageContentType +
            ';base64,' +
            product.image" style="max-height:150px;" alt="product image" />
```

```
</div>
<div class="col col-xs-12">
    <div class="d-flex w-100 justify-content-between">
        ...
        </div>
        ...
        <small>
            ...
            </small>
        </div>
    </div>
</a>
</div>
</div>
```

Take a look at the code highlighted in bold. We added a Bootstrap row (<https://getbootstrap.com/docs/4.0/layout/grid/>) with two-column div tags. The first div tag takes up 2 columns in a 12-column grid specified by `col-2`, while we also say that when the display is `xs` (**extra small**), the div tag should take 12 columns using `col-xs-12`.

The second div tag is kept responsive by specifying `col` so that it takes the remaining available columns after the first div tag, and when the display is extra small, it takes up to 12 columns as well. The image inside the first column div tag uses a data URL as `src` tag to render the image. Now we have an even better view:

## Products

+ Create a new Product

	<b>JHipster T-Shirt-Men</b> JHipster T-Shirt for men <b>Price: 20</b> Size: M	1
	<b>JHipster T-shirt-Women</b> JHipster T-shirt for Women <b>Price: 20</b> Size: S	1

5. We can polish it further. We can use the Angular currency pipe (<https://angular.io/api/common/CurrencyPipe>) for the price and remove the redundant label for it by changing it to `{{product.price | currency: 'USD'}}`. We can add a label for the category shown on the right-hand side of the list as well.
6. Finally, we can add the **Edit** and **Delete** buttons back, but we need to show them only for users who have the `ADMIN` role so that normal users will only be able to view the product listing. We can copy the HTML code for the `edit` and `delete` buttons from the original table. The final code will have the following structure:

```
<div *ngIf="products">
    <div class="list-group">
        <a ...>
            <div class="row">
                ...
                <div class="col col-xs-12">
                    <div class="d-flex w-100 justify-content-between">
                        ...
                        <small *ngIf="product.productCategory">
                            <a [routerLink]=["/product-category",
                                product.productCategory?.id, 'view']">
                                Category: {{ product.productCategory?.id }}
                            </a>
                        </small>
                    </div>
                    <small class="mb-1">{{ product.description }}</small>
                    <p class="mb-1">{{ product.price | currency: 'USD' }}</p>
                    <small>
                        ...
                    </small>
                    <div *jhiHasAnyAuthority="'ROLE_ADMIN'">
                        <button type="submit" [routerLink]=["/product",
                            product.id,
                            'edit']" class="btn btn-primary btn-sm">
                            <fa-icon [icon]="'pencil-alt'"></fa-icon>
                            <span class="d-none d-md-inline">
                                jhiTranslate="entity.action.edit">Edit</span>
                        </button>
                        <button type="submit" (click)="delete(product)">
                            <span class="d-none d-md-inline">
                                <fa-icon [icon]="'times'"></fa-icon>
                                <span class="d-none d-md-inline">
                                    jhiTranslate="entity.action.delete">Delete</span>
                            </span>
                        </button>
                    </div>
                </div>
            </div>
        </a>
    </div>
</div>
```

```
</div>
</div>
</a>
</div>
</div>
```

The `*jhiHasAnyAuthority="'ROLE_ADMIN'"` directive is provided by JHipster and can be used to control the presentation based on user roles. By default, JHipster provides `ROLE_ADMIN` and `ROLE_USER`, but controlling this only on the client-side is not secure as it can be easily bypassed, so we should secure this on the server-side as well. We will look at this later in the chapter. Log out and log in again using the user account to see the directive in action:

The screenshot shows a web application for managing products. At the top left is a red header bar with the text "Products". At the top right is a blue button labeled "+ Create a new Product". Below the header, there are two product cards. The first card is for "JHipster T-Shirt-Men", featuring a blue t-shirt with a cartoon character on it. It includes the text "JHipster T-Shirt for men", the price "\$20.00", and size "Size: M". It has two buttons: a blue "Edit" button and a red "Delete" button. To the right of the card, it says "Category: 1". The second card is for "JHipster T-shirt-Women", also featuring a blue t-shirt with a cartoon character. It includes the text "JHipster T-shirt for Women", the price "\$20.00", and size "Size: S". It has similar "Edit" and "Delete" buttons and a "Category: 1" label.

Now, let's also add the `*jhiHasAnyAuthority="'ROLE_ADMIN'"` directive to the `Create` button element.

## Bringing back the sorting functionality

Now that our view is much better, let's bring back the sorting functionality that we originally had. Since we do not have table headers anymore, we can use some buttons to sort the information based on certain fields that are important.

Let's use the Bootstrap button group (<https://getbootstrap.com/docs/4.0/components/button-group/>) for this. Place the following snippet over the `<div class="list-group">` element that we created earlier:

```
<div class="mb-2 d-flex justify-content-end align-items-center">
  <span class="mx-2 col-1">Sort by</span>
  <div class="btn-group" role="group" jhiSort [(predicate)]="predicate"
    [(ascending)]="reverse" [callback]="transition.bind(this)">
    <button type="button" class="btn btn-light" jhiSortBy="name">
      <span class="d-flex">
        <span jhiTranslate="storeApp.product.name">Name</span>&ampnbsp
        <fa-icon [icon]="'sort'"></fa-icon>
      </span>
    </button>
    <button type="button" class="btn btn-light" jhiSortBy="price">
      <span class="d-flex">
        <span jhiTranslate="storeApp.product.price">Price</span>&ampnbsp
        <fa-icon [icon]="'sort'"></fa-icon>
      </span>
    </button>
    <button type="button" class="btn btn-light" jhiSortBy="size">
      <span class="d-flex">
        <span jhiTranslate="storeApp.product.size">Size</span>&ampnbsp
        <fa-icon [icon]="'sort'"></fa-icon>
      </span>
    </button>
    <button type="button" class="btn btn-light" jhiSortBy="productCategory.id">
      <span class="d-flex">
        <span jhiTranslate="storeApp.product.productCategory">Product
          Category</span>&ampnbsp
        <fa-icon [icon]="'sort'"></fa-icon>
      </span>
    </button>
  </div>
</div>
```

We can use the Bootstrap margin and flexbox utility classes, such as `mb-2 d-flex justify-content-end align-items-center`, to position and align the item properly. We use the `btn-group` class on a `div` element to group our `button` elements together, on which we have placed the `jhiSort` directive and its bound properties, such as `predicate`, `ascending`, and `callback`. On the buttons themselves, we use the `jhiSortBy` directive to specify which field they would use to sort. Now our page looks as follows, where products are sorted by price:

Products

+ Create a new Product

Sort by Name ▲ Price ▲ Size ▲ Product Category ▲

Spring T-shirt-Men Spring T-shirt for men \$10.00 Size: XL <a href="#">Edit</a> <a href="#">Delete</a>	Category: 1
Spring T-shirt Spring T-shirt women's \$15.00 Size: M <a href="#">Edit</a> <a href="#">Delete</a>	Category: 1

## Adding a filtering functionality

Finally, let's add some good old client-side filtering to the page:

1. First, let's add a new instance variable called `filter` of the `string` type to the `ProductComponent` class in the `product.component.ts` file:

```
export class ProductComponent implements OnInit, OnDestroy {  
  
    ...  
    filter: string;  
  
    constructor(  
        ...  
    ) {  
        ...  
    }  
    ...  
}  
}
```



JHipster provides an option to enable server-side filtering using the JPA metamodel. Another option is to enable Elasticsearch, for which JHipster will automatically create full-text search fields for every entity. You should use these for any serious filtering requirements you might have.

2. Now let's use this variable in the `product.component.html` file. Add the highlighted snippet from the following code to the `div` tag we created for the **Sort by** buttons.

```
<div class="mb-2 d-flex justify-content-end align-items-center">
  <span class="mr-2 col-2">Filter by name</span>
  <input type="search" class="form-control" [(ngModel)]="filter">
  <span class="mx-2 col-1">Sort by</span>
  <div class="btn-group" role="group">
    ...
  </div>
</div>
```

We bound the `filter` variable to an input element using the `ngModel` directive, and using `[( )]` ensures two-way binding on the variable.



The `[(ngModel)]="filter"` phrase creates a two-way binding, `[ngModel]="filter"` creates a one-way binding from the model to the view, and `(ngModel)="filter"` creates a one-way binding from the view to the model.

3. Finally, update the `ngFor` directive on our list's group item element as follows. We use a pipe provided by JHipster to filter the list using the `name` field of the product:

```
*ngFor="let product of (products | pureFilter:filter:'name');
trackBy:
  trackId"
```

That's it. We should get a shiny filter option on our screen:

The screenshot shows a product listing interface. At the top, there's a header with 'Products' on the left and a blue button 'Create a new Product' on the right. Below the header, there are search/filter options: 'Filter by name' with a text input containing 'women', 'Sort by' dropdowns for 'Name' (sorted by 'Category'), 'Price' (sorted by 'Price'), 'Size' (sorted by 'Size'), and 'Product Category' (sorted by 'Category'). A large product card is displayed, featuring a blue t-shirt with a cartoon character on it. The card has the title 'JHipster T-shirt-Women', a description 'JHipster T-shirt for Women', a price '\$20.00', and a size 'Size: S'. Below the card are two buttons: 'Edit' (blue) and 'Delete' (red). At the bottom of the page, a message says 'Showing 1 - 4 of 4 items.' followed by a pagination control with buttons for '««', '«', '1', '»', and '»»'.

The UX is much better than before, but for a real-world use case, you could build a much better UI for the client-facing website, with features to add items to a cart, pay for items online, and so on, and leave this part for the back office to use. Let's commit this to Git.

This is very important for managing changes to the project later. Run the following commands:

```
> git add --all  
> git commit -am "update product listing page UI"
```

Our code is now committed to `git`. Let's now see how we can edit an entity using JHipster.

## Editing an entity using the JHipster entity sub-generator

While looking through the generated entity screens, we might realize that there are some minor issues that affect the user experience. For example, on the product screens, we have a relationship with a product category, but when choosing the product category from the drop-down menu during creation or when showing the category in the list, we show the category by its ID, which is not user-friendly.

It would be nice if we could show the product category name instead. This is the default JHipster behavior, but it can be customized while defining the relationships. Let's see how we can make our generated screens more user-friendly by editing the JDL model. This will overwrite existing files, but since we are using git, we can easily cherry-pick the changes we made. We will see how this is done in a moment:

1. In our JDL, we defined relationships between the entities using the following code:

```
relationship OneToOne {
    Customer{user} to User
}

relationship ManyToOne {
    OrderItem{product} to Product
}

relationship OneToMany {
    Customer{order} to ProductOrder{customer},
    ProductOrder{orderItem} to OrderItem{order},
    ProductOrder{invoice} to Invoice{order},
    Invoice{shipment} to Shipment{invoice},
    ProductCategory{product} to Product{productCategory}
}
```

By specifying the field to use for displaying the relationship in JDL using the (`<field name>`) syntax, we can change how the client-side code displays relationships:

```
relationship OneToOne {
    Customer{user(login)} to User
}

relationship ManyToOne {
    OrderItem{product (name)} to Product
}

relationship OneToMany {
    Customer{order} to ProductOrder{customer(email)},
    ProductOrder{orderItem} to OrderItem{order(code)},
    ProductOrder{invoice} to Invoice{order(code)},
    Invoice{shipment} to Shipment{invoice(code)},
    ProductCategory{product} to Product{productCategory(name)}
}
```

- Let's run this using the `import-jdl` command. The command only generates entities that underwent changes from the last run. But before we run, let's also switch to a new branch, because it's good practice to make major changes on a separate branch and merge them back so you have more control:

```
> git checkout -b entity-update-display-name  
> jhipster import-jdl online-store.jdl
```

Accept the changes to the files and wait for the build to finish.



Read more about Git flow at <https://guides.github.com/introduction/flow/>.

- Now let's look at the entity pages to verify that the display names are being used properly and create some entities to try it out. Now we realize that the `Invoice` entity has empty drop-down menus when used in other entities, and that is because the `Invoice` entity does not have a field called `code`. Since we use `{{invoice.order?.code}}` in the template, the `?` symbol makes Angular skip undefined values, preventing errors in rendering.

This is easy to fix. Sometimes, we might want to make a small change to an entity after we have created it using JDL and the `import-jdl` command. The best way would be to make the change in JDL and regenerate it using the `import JDL` command, as we saw in the previous code. There is also another option: the entity sub-generator can yield the same result. For the sake of familiarizing yourself with this option, let's use the entity sub-generator to add the field to our `Invoice` entity:

- Run the following command:

```
> jhipster entity Invoice
```

- From the options, select `Yes`, add more fields and relationships:

```
Using JHipster version installed globally  
Executing jhipster:entity Invoice  
Options:
```

```
Found the .jhipster/Invoice.json configuration file, entity can  
be automatically generated!
```

```
The entity Invoice is being updated.
```

? Do you want to update the entity? This will replace the existing files for this entity, all your custom code will be overwritten

Yes, re generate the entity

➤ Yes, add more fields and relationships

Yes, remove fields and relationships

No, exit

3. Select Yes for the next question and provide the field name, type, and validation in the questions that follow:

Generating field #7

? Do you want to add a field to your entity? Yes

? What is the name of your field? code

? 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

===== Invoice =====

Fields

date (Instant) required

details (String)

status (InvoiceStatus) required

paymentMethod (PaymentMethod) required

paymentDate (Instant) required

paymentAmount (BigDecimal) required

code (String) required

Relationships

shipment (Shipment) one-to-many

order (ProductOrder) many-to-one

Generating field #8

? Do you want to add a field to your entity? (Y/n)

4. Select n for the prompts that follow to add more fields and relationships.  
Accept the proposed file changes, and that's it—we are done.
5. Now, just make sure that you update the JDL so that the Invoice entity has code String required as a field.



You could also run `jhipster export-jdl online-store.jdl` to export the current model back to the JDL. The `export-jdl` command creates a JDL file with all the information about current entities, relationships, and the application you have.

Now that we have displayed entity relationships properly, we also need to make sure certain entities have mandatory relationship values. For example, consider the following:

- For customers, it should be mandatory to have a user.
- `ProductOrder` should have a customer.
- Order items should have an order.
- An invoice should have an order.
- The shipment should have an invoice.

Since JHipster supports making relationships that are required, we can make these changes using JDL. Update the relationships to the following snippet in `online-store.jdl`:

```
relationship OneToOne {  
    Customer{user(login) required} to User  
}  
  
relationship ManyToOne {  
    OrderItem{product(name) required} to Product  
}  
  
relationship OneToMany {  
    Customer{order} to ProductOrder{customer(email) required},  
    ProductOrder{orderItem} to OrderItem{order(code) required},  
    ProductOrder{invoice} to Invoice{order(code) required},  
    Invoice{shipment} to Shipment{invoice(code) required},  
    ProductCategory{product} to Product{productCategory(name)}  
}
```

Now, run `jhipster import-jdl online-store.jdl` and accept the proposed updates. Make sure that you check what has changed using the `git diff` command or your Git UI tool.

Let's commit this step so that it can be rolled back if required:

```
> git add --all  
> git commit -am "entity relationships display names and required update"
```

Now we have a problem: regenerating the entities overwrote all the files, and that means we lost all the changes we made for the product listing page, but since we are using Git, it's easy to get it back. So far, our project has only a few commits, so it will be easy to cherry-pick the commit we made for the product listing UI change and apply it back on top of the current code base; however, in real-world scenarios, there could be a lot of changes before you can regenerate the JDL, and so it will require some effort to verify and merge the required changes back. Always rely on pull requests so that you can see what has changed and others can review and find any issues.

Let's cherry-pick the changes that we need.



Refer to the documentation for cherry-picking advanced options at <https://git-scm.com/docs/git-cherry-pick>.

Since the commit we need is the last one on the master, we can simply use `git cherry-pick master`. We could also switch to the master and use the `git log` command to list the commits, then copy the commit hash of the required commit and use that with `git cherry-pick <commit-sha>`.

This results in merge conflicts, as the `product.component.html` file was updated in the commit that we picked on our current branch tip. We need the incoming change from the commit, but we also need to update the product category display name from ID to code, so let's accept the incoming change and make a manual update from

`{product.productCategory?.id}` to `{product.productCategory?.name}`.

Resolve the conflict by staging the file and commit. Now we can merge the branch into the master:

```
> git cherry-pick master
// Fix merge conflict
> git add src/main/webapp/app/entities/product/product.component.html
> git commit -am "cherrypick: update product listing page UI"
> git checkout master
> git merge --no-ff entity-update-display-name
```

And now everything is merged.



If you are new to Git, it is advisable to use a UI tool such as SourceTree or GitKraken to cherry-pick and resolve merge conflicts. IDEs such as IntelliJ and editors such as VS Code also provide good options for these.

Now our page view should be good:

Product	Description	Price	Size	Category
JHipster T-Shirt-Men	JHipster T-Shirt for men	\$20.00	M	T-Shirt
JHipster T-shirt-Women	JHipster T-shirt for Women	\$20.00	S	T-Shirt

Of course, we could also make it more user-friendly by making the product listing our home page. But for now, let's skip that.



Adding validations to the relationships might cause issues with fake data being generated, and so at this point, you might have to fix the unique constraint violations in the generated fake data CSV files under `src/main/resources/config/liquibase/fake-data/` or disable the ability to load fake data by removing the `faker` profile in the `spring.liquibase.contexts` property in the application configuration, `src/main/resources/config/application-dev.yml`.

Since we were working on the client-side code, we didn't pay attention to the server-side code that was changed while we were doing this. We need to compile the Java code to reload our server. Let's run `./gradlew compileJava`.

Unfortunately, we receive an error during the reload regarding a failure to update the database changelogs by Liquibase due to a checksum error:

```
liquibase.AsyncSpringLiquibase : Liquibase could not start correctly, your
database is NOT ready: Validation Failed:
    5 change sets check sum
config/liquibase/changelog/20180114123500_added_entity_Customer.xml::201801
14123500-1::jhipster was: 7:3e0637bae010a31ecb3416d07e41b621 but is now:
```

```
7:01f8e1965f0f48d255f613e7fb977628
config/liquibase/changelog/20180114123501_added_entity_ProductOrder.xml::20
180114123501-1::jhipster was: 7:0ff4ce77d65d6ab36f27b229b28e0cda but is
now: 7:e5093e300c347aacf09284b817dc31f1
config/liquibase/changelog/20180114123502_added_entity_OrderItem.xml::20180
114123502-1::jhipster was: 7:2b3d9492d127add80003e2f7723903bf but is now:
7:4beb407d4411d250da2cc2f1d84dc025
config/liquibase/changelog/20180114123503_added_entity_Invoice.xml::2018011
4123503-1::jhipster was: 7:5afaca031815e037cad23f0a0f5515d6 but is now:
7:fadec7bfabcd82dfc1ed22c0ba6c6406
config/liquibase/changelog/20180114123504_added_entity_Shipment.xml::201801
14123504-1::jhipster was: 7:74d9167f5da06d3dc072954b1487e11d but is now:
7:0b1b20dd4e3a38f7410b6b3c81e224fd
```

This is because of the changes that were made to the original changelog by JHipster. In an ideal world, new schema changes should be made in new changelogs so that Liquibase can apply them, but JHipster doesn't generate this by default yet. For local development using an H2 DB, we can run `./gradlew clean` to clear the DB and start the application again. But in real use cases, you might be using an actual DB, and you would want to retain the data, so we would have to handle this manually here using the `diff` features provided by Liquibase.

JHipster provides integration for Liquibase in both Gradle and Maven builds. You can make use of it to create new changelogs and to create diff changelogs. In cases like these, when we would like to resolve conflicts while retaining data, the Liquibase diff feature is our friend. With Gradle, you could run the `./gradlew liquibaseDiffChangeLog` command to create a diff changelog of your changesets and the database. You can add this changeset to the `src/main/resources/config/liquibase/master.xml` file and it will be applied the next time you restart your server.

By default, the command is configured to run against your development database. If you would like to do this against your production database, then just update the `liquibase` command definition in the `build.gradle` file with the details of the production DB. Refer to <http://www.jhipster.tech/development/#using-a-database> for more details.



If you want to clear checksums in your DB, use the  
`./gradlew liquibaseClearChecksums` task.

# Changing the look and feel of the application

The good thing about using Bootstrap is that it lets us easily change the look and feel of the application using any available Bootstrap themes. Let's see how we can install a cool theme for our application, and then fine-tune the styles to fit our needs using SASS variables provided by Bootstrap.

There are hundreds of Bootstrap themes out there. Since we are using Bootstrap 4, it is important to pick a theme that is made for Bootstrap 4.



Bootswatch is a nice collection of themes for Bootstrap. JHipster has an option to select a Bootswatch theme during application generation. You can see all the available themes at <https://bootswatch.com/>.

Let's use a Bootswatch theme called **materia**. You can skip adding the dependencies using the following steps if you choose materia as the default theme during application generation:

1. In your Terminal, run `npm install --save bootswatch` to install all the themes. Don't worry; we will only import the theme that we want to use, so you do not have to worry about installing all of the themes.
2. Now, let's import this using SASS. Open `src/main/webapp/content/scss/vendor.scss`, find the line `@import '~bootstrap/scss/bootstrap';`, and add the following code highlighted in bold:

```
// Override Boostrap variables
@import "bootstrap-variables";
@import '~bootswatch/dist/materia/variables';
// Import Bootstrap source files from node_modules
@import '~bootstrap/scss/bootstrap';
@import "~bootswatch/dist/materia/bootswatch";
```

The name of the theme here is **materia**, but you can use any theme available in Bootswatch here. Make sure that the name is all in lowercase. Also, note the order of the imports. It is important that we import the theme variables after importing the Bootstrap variables and that we import the themes after importing the Bootstrap theme so that the SASS variables and styles are overridden properly.

We can customize the theme further by overriding the Bootstrap variables defined in `src/main/webapp/content/scss/_bootstrap-variables.scss`.

You can override any variable supported by Bootstrap. The full list of supported variables can be found in `node_modules/bootstrap/scss/_variables.scss`.

For example, let's change some colors in `_bootstrap-variables.scss`, as follows:

```
$primary: #032b4e;
$success: #1df54f;
$info: #17a2b8;
$warning: #ffc107;
$danger: #fa1a30;
```

There might be some UI glitches when you apply a new theme; you could solve them by updating the generated SASS files.

For example, you can add custom CSS/SASS to `src/main/webapp/content/scss/global.scss` to fix UI glitches in themes or to add new global UI styles.

We now have a cool new theme:

NAME	PRICE	SIZE	PRODUCT CATEGORY
JHipster T-Shirt-Men	\$20.00	M	T-Shirt
JHipster T-shirt-Women	\$20.00	S	T-Shirt

More information can be found at <https://getbootstrap.com/docs/4.0/getting-started/theming/>.

3. Let's commit this:

```
> git add --all  
> git commit -am "update bootstrap theme using bootswatch"
```

Let's now learn how to add additional languages.

## Adding a new i18n language

Since we enabled i18n support for our application, we can add new i18n languages easily at any time using the JHipster language generator. Let's add a new language to our application:

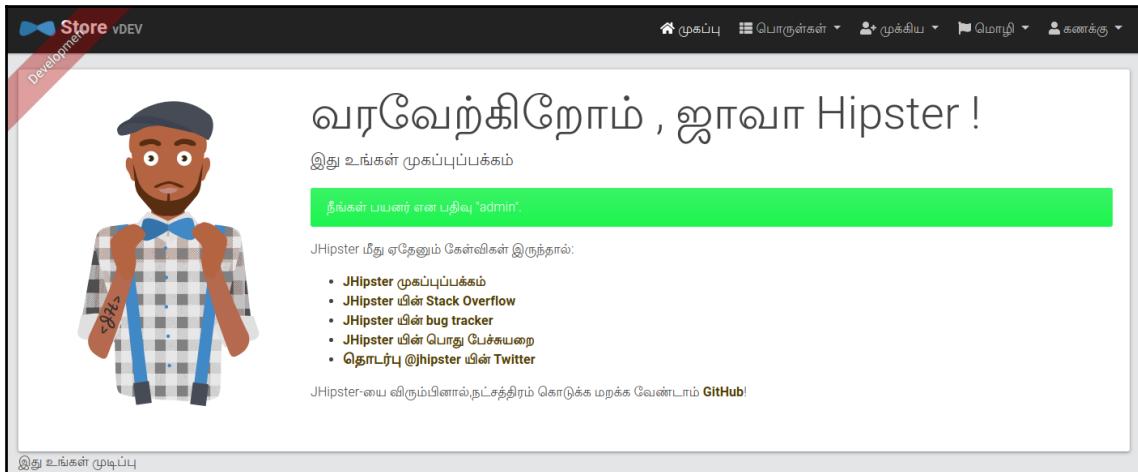
1. In the Terminal, switch to a new branch and run the following command:

```
> git checkout -b add-tamil-language  
> jhipster languages
```

2. You will now see a prompt like the following, where you can choose any available language listed:

```
Using JHipster version installed locally in current project's  
node_modules  
Executing jhipster:languages  
Options:  
  
Languages configuration is starting  
? Please choose additional languages to install (Press <space> to  
select, <a> to toggle all, <i> to inverse selection)  
>○ Arabic (Libya)  
○ Armenian  
○ Catalan  
○ Chinese (Simplified)  
○ Chinese (Traditional)  
○ Czech  
○ Danish  
(Move up and down to reveal more choices)
```

3. Let's select **Tamil** here for now. Accept the proposed file changes and you are good to go. You will have to restart the `npm start` command. Once the application refreshes, you can see the new language in the language drop-down menu in the application menu. Now, wasn't that easy? Have a look at the following screenshot:



But there is a problem. Since we have some entities and we added a new language, we will need to get i18n Tamil files for entities as well. We can do this easily by running the `jhipster --with-entities` command, which will regenerate the application along with the entities. Make sure that you carefully stage only the changes that you need (i18n-related JSON files) from the diff and discard the remaining changes. The following are the files and folders that need to be staged:

```
.yo-rc.json
src/main/resources/i18n/messages_ta.properties
src/test/resources/i18n/messages_ta.properties
src/test/java/com/mycompany/store/service/MailServiceIT.java
src/main/webapp/app/shared/language/find-language-from-key.pipe.ts
src/main/webapp/app/shared/language/language.constants.ts
webpack/webpack.common.js
webpack/webpack.prod.js
src/main/webapp/i18n/ta/*
```

Now, let's commit this and merge it back to the master. If we have picked only i18n-related changes, then we shouldn't have any merge conflicts:

```
> git add --all
> git commit -am "add Tamil as additional language"
```

```
> git checkout master  
> git merge --no-ff add-tamil-language
```

Now, let's see how to set up authorization for the application.

## Authorization with Spring Security

As you may have noticed, when it comes to generated code, JHipster doesn't provide much in terms of role-based security, authorization management, and so on. This is intentional, as these heavily depend on the use case and are most often associated with the business logic of the application. So, it would be better if these features were hand-coded by the developers as part of the business code.

Normal users have `ROLE_USER` and admin users have `ROLE_ADMIN` assigned in their user management. For our use case, there are a few security holes that we need to take care of:

- Normal users should only have access to view the product listing, product orders, order items, invoices, and shipments.
- Normal users should not have access to create/edit/delete entities via the CRUD API.
- Normal users should not be able to access the product orders, order items, invoices, and shipments of other users.

We could overcome these issues using features provided by Spring Security.

## Limiting access to entities

First, let's limit the access for normal users. This can be done easily at the API level using Spring Security. Add the following snippet to the `configure` method of `src/main/java/com/mycompany/store/config/SecurityConfiguration.java`.

Add it right before the line `.antMatchers("/api/**").authenticated()`. The position is very important:

```
.antMatchers("/api/customers").hasAuthority(AuthoritiesConstants.ADMIN)  
.antMatchers("/api/product-  
categories").hasAuthority(AuthoritiesConstants.ADMIN)
```

We specify that when the request path matches `api/customers` or `api/product-categories`, the user should have `ROLE_ADMIN` to access them. Now recompile with `./gradlew compileJava`, then sign out and log in as `user` and try to access the customer entity page. Look at the console in your browser's development tools and you should see a 403 Forbidden error for calls made to `GET http://localhost:9000/api/customers`.

Now that our backend handles this properly, let's hide these entries in the menu for normal users. Let's add a `*jhiHasAnyAuthority="'ROLE_ADMIN'"` directive to the list elements for the customer and product categories in `src/main/webapp/app/layouts/navbar/navbar.component.html`.

Now only admin users will see these items on the menu.

## Limiting access to create/edit/delete entities

Now we need to ensure that only admin users can edit entities; normal users should only be able to view entities authorized to them. For this, it would be better to handle it at the API level using the Spring Security `PreAuthorize` annotation. Let's start with the `OrderItem` entity. Go to

`src/main/java/com/mycompany/store/web/rest/OrderItemResource.java` and add `@PreAuthorize("hasAuthority('ROLE_ADMIN')")` to the `createOrderItem`, `updateOrderItem`, and `deleteOrderItem` methods:

```
@DeleteMapping("/order-items/{id}")
@PreAuthorize("hasAuthority('ROLE_ADMIN')")
public ResponseEntity<Void> deleteOrderItem(@PathVariable Long id) {
    ...
}
```

We are asking Spring Security interceptors to provide access to these methods only when the user has `ROLE_ADMIN`. The `PreAuthorize` annotation stops access before executing the method. Spring Security also provides `PostAuthorize` and more general `Secured` annotations. More information about these can be found in the Spring Security documentation at <https://projects.spring.io/spring-security/>.

Compile the backend using `./gradlew compileJava` or the IDE. Now go to the order items page and try to create an order item. You will get a POST

`http://localhost:9000/api/order-items` 403 (Forbidden) error from the API call on the web console. Now, let's add the annotation to all the entity Resource class's create, update, and delete methods. You could skip customer and product category entities, as they are entirely forbidden to `ROLE_USER` already.

Let's also hide the create, edit, and delete buttons from the Angular views using the `*jhiHasAnyAuthority="'ROLE_ADMIN'"` directive.

## Limiting access to the data of other users

Now, this is a little more tricky, as this requires us to change code at the service layer on the backend, but it is not hard. Let's get right to it.

Let's start with the product order entity. Modify the `findAll` method in `src/main/java/com/mycompany/store/service/ProductOrderService.java` as follows:

```
@Transactional(readOnly = true)
public Page<ProductOrder> findAll(Pageable pageable) {
    log.debug("Request to get all ProductOrders");
    if (SecurityUtils.isCurrentUserInRole(
        AuthoritiesConstants.ADMIN)) {
        return productOrderRepository.findAll(pageable);
    } else
        return productOrderRepository.findAllByCustomerUserLogin(
            SecurityUtils.getCurrentUserLogin().get(),
            pageable
        );
}
```

As you can see, we modified the original call to `productOrderRepository.findAll(pageable)` so that we call it only when the current user has the Admin role; otherwise, we call `findAllByCustomerUserLogin`, but our generated `ProductOrderRepository` interface does not have this method yet, so let's add that. In `src/main/java/com/mycompany/store/repository/ProductOrderRepository.java`, let's add a new method as follows. Currently, the interface doesn't have any methods and only uses methods inherited from `JpaRepository`:

```
Page<ProductOrder> findAllByCustomerUserLogin(String login, Pageable pageable);
```

There is a lot of magic going on here. This is a Spring Data interface, and so we can simply write a new method and expect Spring Data to create an implementation for this automatically; we just need to follow the naming conventions. In our use case, we need to find all product orders where the user relationship for the customer has the same login as our current logged in user. In SQL, this would be as follows:

```
select * from product_order po cross join customer c cross join jhi_user u
where po.customer_id=c.id and c.user_id=u.id and u.login=:login
```

In simple terms, we could say *find all product orders where customer.user.login equals login*, and that is exactly what we have written as the `findAllByCustomerUserLogin` method. The entity under operation is implicit, and so the product order is omitted. By providing the `Pageable` parameter, we tell Spring Data to provide us a page from the paginated list of entities. You can refer to the Spring Data documentation at <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/> for more information.

While calling the `productOrderRepository.findAllByCustomerUserLogin` method, we can pass the current user login using the `SecurityUtils.getCurrentUserLogin()` method. The `SecurityUtils` class is generated by JHipster as well, as it has useful methods such as `getCurrentUserLogin`, `getCurrentUserJWT`, `isAuthenticated`, and `isCurrentUserInRole`.

That's it. Now log in as `admin` and create two new users and two customers, and create product orders for each of them. Then log out and log in again as the default user and see whether you can see the product order for one of the newly created users.

Now let's make similar updates for the other services. Note that each method is slightly different based on the relationships between the entities. The repository methods for these would be as follows:

For `src/main/java/com/mycompany/store/repository/InvoiceRepository`:

```
Page<Invoice> findAllByOrderCustomerUserLogin(String login, Pageable
pageable);
```

For `src/main/java/com/mycompany/store/repository/OrderItemRepository`:

```
Page<OrderItem> findAllByOrderCustomerUserLogin(String login, Pageable
pageable);
```

For `src/main/java/com/mycompany/store/repository/ShipmentRepository`:

```
Page<Shipment> findAllByInvoiceOrderCustomerUserLogin(String login,
Pageable pageable);
```

Now we need to make similar changes to the `findOne` method on the services.

For `ProductOrderService`, these would be as follows:

```
@Transactional(readOnly = true)
public ProductOrder findOne(Long id) {
    log.debug("Request to get ProductOrder : {}", id);
    if (SecurityUtils.isCurrentUserInRole(
        AuthoritiesConstants.ADMIN)) {
        return productOrderRepository.findById(id);
    } else
        return productOrderRepository.
            findOneByIdAndCustomerUserLogin(
                id,
                SecurityUtils.getCurrentUserLogin().get()
            );
}
```

As you can see, we changed the methods to find one by its ID and customer user login. The repository method for this would be as follows:

```
Optional<ProductOrder> findOneByIdAndCustomerUserLogin(Long id, String
login);
```

For `src/main/java/com/mycompany/store/repository/InvoiceRepository`:

```
Optional<Invoice> findOneByIdAndOrderCustomerUserLogin(Long id, String
login);
```

For `src/main/java/com/mycompany/store/repository/OrderItemRepository`:

```
Optional<OrderItem> findOneByIdAndOrderCustomerUserLogin(Long id, String
login);
```

For `src/main/java/com/mycompany/store/repository/ShipmentRepository`:

```
Optional<Shipment> findOneByIdAndInvoiceOrderCustomerUserLogin(Long id,
String login);
```

The same queries can also be written using the `@Query` annotation provided by Spring Data.

Now change the `findOne` method for other entities as well, and that's it. We have implemented a good role-based authorization logic for the application.

Let's commit this checkpoint:

```
> git add --all  
> git commit -am "update role based authorization logic"
```

In a real-world scenario, the changes we have made so far will not be enough for an e-commerce website. But since our aim is to learn JHipster and its supported tools rather than to create a feature-perfect application, consider this a minimum-viable product. To make this e-commerce application usable, we would need to build more features, such as a shopping cart, invoice generation, customer registration, and so on. Why don't you take these exercises up as assignments and see whether you can build more features for this application? This would be part of the next steps to take once you finish the book. The use case and instructions will be described in Chapter 15, *Best Practices with JHipster*.

## Summary

In this chapter, we saw how we can easily customize a web application created using JHipster. We also learned about Angular and Bootstrap when we customized our product listing page. In addition to this, we saw how to secure our application with role-based authorization using Spring Security. We also learned about Spring Data and used Git to manage our source code properly. We saw our application evolving with business logic and becoming more user-friendly. In the next chapter, we will see how we can integrate continuous integration with our application using Jenkins.

# 3

## Section 3: Continuous Integration and Testing

This section begins by using Jenkins to test and set up a continuous integration pipeline and also build and package apps using Docker. By the end of this section, you will be introduced to cloud deployment options that are supported by JHipster.

This section comprises the following chapters:

- Chapter 6, *Testing and Continuous Integration*
- Chapter 7, *Going into Production*

# 6

# Testing and Continuous Integration

Now that we have scaffolded and developed our e-commerce application, it's time to make it ready for deployment to our production environment. Before that, there are two important aspects of software engineering that we need to look at, *quality* and *stability*. In this chapter, we will see how this can be achieved using modern DevOps practices, such as CI and automated testing. With this, you will be able to set up your own Jenkins CI pipeline and will get a basic introduction to **continuous integration/continuous deployment (CI/CD)**.

We will also explore the following:

- Fixing and running tests
- CI/CD tools
- Setting up CI with Jenkins using the JHipster CI-CD sub-generator



**DevOps** is a software engineering practice that unifies software **development (Dev)** and software **operations (Ops)**. The main focus of DevOps is automation and monitoring at all stages of software engineering, such as development, integration, testing, deployment, and infrastructure management. DevOps is one of the most trending engineering practices of this decade, and CI and CD are two of its core aspects.

First, let's see how to fix our tests that would have been broken when we updated the code in the previous chapter.

# Fixing and running tests

JHipster generates different types of tests for an application; some are optional and can be enabled while creating a new application, while some are always generated. The following are the different types of tests supported by JHipster and that we already talked about in Chapter 2, *Getting Started with JHipster*:

- **Server-side unit tests:** These are test cases for individual classes and methods and are done using JUnit.
- **Server-side integration tests:** These perform tests on the Spring components (REST controllers and services) making use of all of the layers behind them. It is done with Spring Test Framework and JUnit.
- **Client-side unit tests:** These are test cases for the client-side components and services and are done using the Jest framework.
- **Client-side end-to-end tests:** The end-to-end tests are written using the Protractor framework and they perform user behavior simulation on the GUI in a real browser. This is optional and can be enabled if required.
- **Performance tests:** JHipster can generate performance and load test specifications for the APIs using Gatling. This is optional and can be enabled if required.
- **Behavior-driven development (BDD) tests:** JHipster can generate BDD tests specifications using Cucumber. This is optional and can be enabled if required.

Before we dive into CI tools, let's first make sure that our tests are working, and there are no failed tests after the changes we made in the previous chapter. In an ideal world, where software development is done using practices such as **TDD** (short for **test-driven development**), writing and fixing tests is done along with the development of the code, and specifications are written before you develop the actual code.

You should try to follow this practice so that you write **failing tests** first for an expected result, and then develop code that will make the tests pass. Since our tests were autogenerated by JHipster, we can at least make sure that they are working when we make changes to the generated code.



JHipster can also generate performance tests using Gatling for the entities. It is very useful, and a must if you are developing a high-availability and high-volume website. This can be enabled when creating the application. See <http://www.jhipster.tech/running-tests/> for more details.

Let's run our unit and integration tests to see whether any of them fail:

1. Head over to your Terminal and navigate to the `online-store` folder first.
2. Let's first run the server-side tests using Gradle:

```
> ./gradlew test integrationTest
```



Note that JHipster generates both unit tests and integration tests for the server-side and these are located in the same package. The unit tests, files named `*Test.java`, are simple JUnit tests intended for unit testing functions and these can be run using the `./gradlew test` command. The integration tests, files named `*IT.java`, are intended for testing a Spring component using the entire Spring environment. They are run with the `SpringBootTest` class and they normally start up the Spring environment, configure all of the required beans, and run the test. These can be run using the `./gradlew integrationTest` command.

Some of our tests failed with the following error trace:

```
com.mycompany.store.web.rest.OrderItemResourceIT >
getAllOrderItems() FAILED
    java.lang.AssertionError at OrderItemResourceIT.java:261

com.mycompany.store.web.rest.OrderItemResourceIT > getOrderItem()
FAILED
    java.lang.AssertionError at OrderItemResourceIT.java:277

com.mycompany.store.web.rest.InvoiceResourceIT > getInvoice()
FAILED
    java.lang.AssertionError at InvoiceResourceIT.java:341

com.mycompany.store.web.rest.InvoiceResourceIT > getAllInvoices()
FAILED
    java.lang.AssertionError at InvoiceResourceIT.java:321

com.mycompany.store.web.rest.ProductOrderResourceIT >
getProductOrder() FAILED
    java.lang.AssertionError at ProductOrderResourceIT.java:257

com.mycompany.store.web.rest.ProductOrderResourceIT >
getAllProductOrders() FAILED
    java.lang.AssertionError at ProductOrderResourceIT.java:241

com.mycompany.store.web.rest.ShipmentResourceIT > getAllShipments()
FAILED
    java.lang.AssertionError at ShipmentResourceIT.java:204
```

```
com.mycompany.store.web.rest.ShipmentResourceIT > getShipment()
FAILED
    java.lang.AssertionError at ShipmentResourceIT.java:220

194 tests completed, 8 failed
```



You could also run the tests from your IDE so that you have a better error message and failure report. Select the entire `src/test` folder, right-click, and select **Run all tests**.

3. These are expected to fail as we changed the Resource classes for these entities in the previous chapter to handle authorizations, and the failure means that it's working perfectly. Fortunately, it's not difficult to fix the tests using Spring. We can use the `@WithMockUser` annotation provided by the Spring test context to provide a mock user for our tests. Add the annotation with user details, as highlighted in the following code, to all of the failing test classes:

```
@SpringBootTest(classes = StoreApp.class)
@WithMockUser(username="admin", authorities={"ROLE_ADMIN"},
    password = "admin")
public class InvoiceResourceIT {
    ...
}
```

4. We are providing a mock user with the `ADMIN` role here. Add it to `OrderItemResourceIT`, `ProductOrderResourceIT`, and `ShipmentResourceIT`. Run the tests again and they should pass.
5. Commit the changes made by running `git commit -am "fix server-side tests with mockUser"`.
6. Now, let's make sure our client-side Jest unit tests are working. Since we didn't make any logic changes on the client-side there shouldn't be any failures. Run the following command:

```
> npm test
```

7. All tests should pass. Let's head over to

```
src/test/javascript/spec/app/entities/product/product.component.spec.ts. We use the Jest framework for our tests. The existing test has the following structure. The beforeEach block sets up the Angular TestBed interface:  
...  
  
describe('Component Tests', () => {  
  describe('Product Management Component', () => {  
    ...  
    beforeEach(() => {  
      TestBed.configureTestingModule({  
        ...  
      })  
      .overrideTemplate(ProductComponent, '')  
      .compileComponents();  
    ...  
  });  
  
  it('Should call load all on init', () => {  
    ...  
  });  
  ...  
});  
});
```

8. Now, let's make sure our Protractor e2e tests are working. Run the following commands in two separate Terminals. Start the server first. Let's clear the database as well by running a clean task so that tests run on a fresh setup:

```
> ./gradlew clean bootRun
```

9. Now, run the e2e tests:

```
> npm run e2e
```

If you prefer not to run scripts via npm or Yarn, you could also run them via Gradle using the node integration provided by JHipster. For example, instead of `npm run e2e`, you could run `./gradlew npm_run_e2e`, and instead of `npm test`, you could run `./gradlew npm_test`. This is useful if you do not want to install Node.js and npm and want everything to be managed for you by Gradle. If you choose Maven instead of Gradle, the same feature is available for that as well.



10. We have two failed tests in the Product entity pages as we made changes there.  
Let's see how we can fix them:

1. Let's open

```
src/test/javascript/e2e/entities/product/product.spec.ts.
```

2. The first failure is with the `should create and save Products` spec; if you look at the code, there is a call to a `productComponentsPage.countDeleteButtons()` method, and, if you inspect the method, it uses a query selector to find the delete buttons on the page. We need to update the query to match the new HTML structure. Find the following line:

```
deleteButtons = element.all(by.css('jhi-product div table .btn-danger'));
```

Change it to this:

```
deleteButtons = element.all(by.css('jhi-product div.list-group a .btn-danger'));
```

3. The next failure is also due to the same reason and the preceding fix should be enough.
11. If you look at the generated e2e tests, for example,  
`src/test/javascript/e2e/entities/customer.spec.ts`, you will see that some tests are commented out. These tests are commented out during generation if an entity has a required relationship field, as we would have to create a relationship first and set its value for the test to work, or if there are validations that JHipster cannot work out. Let's focus on only the Customer page test. Uncomment the test named `should create and save Customers` and change the `describe` function to `describe.only` on the file so that only this test file is executed:

```
describe.only('Customer e2e test', () => {  
  ...  
});
```

12. First, uncomment the commented out imports. Now, execute `npm run e2e` and we should see one failing test. First, let's fix the email field by providing a valid email format:

```
it('should create and save Customers', () => {
```

```
    ...
    customerUpdatePage.setEmailInput('email@email.com');
    ...
    expect(customerUpdatePage.getEmailInput())
      .toEqual('email@email.com');
    ...
});
```

13. Run `npm run e2e` again and this time it should pass. But since we have a one-to-one relationship between the user and customer, the test will fail if we run it again; hence, we need to delete the row created after it. Let's uncomment the test case for the delete action. Uncomment the `customerDeleteDialog` variable at the beginning of the spec as well. Before running the tests again, we would have to manually clear the last entry created by the failed test to avoid unique constraint violations.
14. Now, run `npm run e2e` twice to confirm this works. Do not forget to change `describe.only` to `describe` on the file so that all tests get executed. Congratulations! You have updated your first Protractor e2e tests.
15. Similarly, fix the commented out e2e tests in other files under `src/test/javascript/e2e/entities` as well. This is a part of the next step's assignment. In some test cases, you would have to create an entity in another module to provide relationships.

Don't forget to commit your changes with `git commit -am "fix client-side e2e tests"`.

Now, let's see what CI is all about.

## Briefing on CI

Having automated testing ensures that we are creating bug-free code and that there are no regressions introduced from the new code. JHipster helps, to an extent, by creating unit and integration tests for the generated code, but, in real use cases, it won't be sufficient. We would have to add server-side unit tests for the business logic that we introduce and integration tests for new APIs we add. You will also have to add more unit tests for business logic handled on the client-side and e2e tests, as JHipster only generates a few sample tests for you and doesn't know anything about your business logic.



The more tests you have, the more confident you will be in changing code, with fewer chances of regression.

Testing and CI is an integral part of full stack development and is an important aspect of DevOps. Testing should be considered as important as developing features to build a quality product. CI is nothing more than continuously merging and testing your new code changes in an isolated environment against your master/main/stable code base to identify potential bugs and regression. It is achieved by running automated unit, integration, end-to-end, and other test suites against the code. For example, if you are working with Git, these are typically run for every commit you make to your master branch and/or for every pull request you create.

Once we have automated tests, we can make use of CI practices to make sure that any new code we introduce doesn't cause any regression in our stable code base. This will give us the confidence to merge new code and deploy that to production.

Modern DevOps teams often go a step further and do continuous delivery (continuous integration and continuous deployment). They often define CI/CD pipelines, which continuously integrate, test, and deploy code to production in a fully automated way.



Teams with good CI and CD setup can deliver more features more frequently with fewer bugs.

Have I stressed the importance of CI enough?

Now, let's take a look at the different CI/CD tools supported by JHipster.

## CI/CD tools

JHipster provides excellent support for the well-known CI/CD tools. Let's take a look at the options available first.

## Jenkins

Jenkins (<https://jenkins.io/>) is one of the leading CI/CD tools out there. It is free and open source. It is an automation server written in Java and supports integration with various version control tools, such as Git, CVS, and SVN. Jenkins has a huge plugin ecosystem and this makes it one of the most flexible platforms. Jenkins can be used for building projects, running automated tests, automating deployment, and so on. It is available as an executable binary for various platforms and as Docker images. Blue Ocean is the latest UI interface for Jenkins, giving it a much-needed breath of fresh air. Jenkins has the concept of a pipeline, achieved by using multiple plugins and a Groovy DSL to define the CI/CD pipeline. Jenkins pipeline plugins provide a comprehensive DSL-based configuration that can be defined in a file called `Jenkinsfile`.

## Azure Pipelines

Azure DevOps Pipelines (<https://azure.microsoft.com/en-us/services/devops/pipelines/>) is a hosted **PaaS** (short for **Platform as a Service**) solution for CI/CD from Microsoft. It is free for public/OSS projects and needs a subscription for use by private/enterprise projects. It is integrated well with GitHub and provides a nice intuitive GUI to work with. It supports applications written in a variety of languages and platforms and is heavily used by open source projects, including JHipster, for their CI needs. It is very easy to set up and use and has a simple YAML-based configuration. Advanced setups are typically done using shell scripts that can be triggered by the YAML configuration using hooks.

## Travis CI

Travis CI (<https://travis-ci.org/>) is an open source hosted **PaaS** solution for CI/CD. It is free for public/OSS projects and needs a subscription for use by private/enterprise projects. It supports applications written in a variety of languages and platforms and is heavily used by open source projects, including JHipster, for their CI needs. It has excellent integration with version control tools and offers an enterprise version as well. It is very easy to set up and use and has a simple YAML-based configuration. Advanced setups are typically done using shell scripts that can be triggered by the YAML configuration using hooks.

## GitLab CI

GitLab CI (<https://about.gitlab.com/features/gitlab-ci-cd/>) is a CI/CD solution available as part of GitLab, a web UI on top of Git. It is well integrated into the platform and is an excellent choice when using GitLab. It is free and open source for use by public projects and has an enterprise version as well. It has both a hosted solution and binaries to be used on-premises.

## GitHub Actions

GitHub Actions (<https://github.com/features/actions>) is the CI/CD solution provided by GitHub, a web UI on top of Git. It is well integrated into the GitHub platform and is a great choice when using GitHub as your source code management solution. It is free for public repositories on GitHub and has a free usage tier for private repositories as well. It supports configuration by YAML like many of the other solutions on the market.

## Setting up Jenkins

Let's use Jenkins as the CI tool for our application.



If you are already familiar with Docker, you can use the official Docker image provided by Jenkins and skip the following steps. The Docker image will be automatically generated by JHipster when creating the CD/CI pipeline in the following section. Visit <http://www.jhipster.tech/setting-up-ci-jenkins2/> for more details.

We first need to set up a local Jenkins instance:

1. Let's download the latest binary from <http://mirrors.jenkins.io/war-stable/latest/jenkins.war>.
2. Now, open a Terminal and navigate to the folder where the file was downloaded.
3. Execute `java -jar jenkins.war --httpPort=8989` from the Terminal to start a Jenkins server. The port should not conflict with our application port. The default password will be printed on the console. Make a copy of it.
4. Navigate to <https://localhost:8989> and paste the password copied before.
5. Click on the **Install suggested plugins** button on the next page and wait for the plugin installation to complete.
6. Create an admin user on the next page and complete the setup.

Now that our Jenkins server is ready, let's go ahead and create a Jenkins pipeline for our project.

## Creating a Jenkins pipeline using JHipster

We can create `Jenkinsfile` for our project using the `ci-cd` sub-generator from JHipster:

1. In a Terminal, navigate to the `online-store` folder first. Now run the following command:

```
> jhipster ci-cd
```

2. You will be asked to select from a list of options, as follows:

```
Welcome to the JHipster CI/CD Sub-Generator
? What CI/CD pipeline do you want to generate? (Press <space> to
select, <a> to toggle all, <i> to inverse selection)
>O Jenkins pipeline
  O Azure Pipelines
  O Travis CI
  O GitHub CI
  O GitLab CI
```

3. Let's select Jenkins pipeline from it. Next, we can choose to run the build inside a Docker container. Let's choose No here:

```
Welcome to the JHipster CI/CD Sub-Generator
? What CI/CD pipeline do you want to generate? Jenkins pipeline
? Would you like to perform the build in a Docker container? (y/N)
```

4. The next question will be about sending build status to GitLab—choose No for that as well:

```
? What CI/CD pipeline do you want to generate? Jenkins pipeline
? Would you like to perform the build in a Docker container? No
? Would you like to send build status to GitLab? No
```

5. Next, we will have an option to choose additional stages. Let's choose the Deploy to \*Heroku\* option here to automatically deploy to Heroku from our CI/CD pipeline:

```
? What CI/CD pipeline do you want to generate? Jenkins pipeline
? Would you like to perform the build in a Docker container ? No
? Would you like to send build status to GitLab ? No
? What tasks/integrations do you want to include ? (Press <space>
to select, <a> to
toggle all, <i> to invert selection)
> Deploy your application to an *Artifactory*
 Analyze your code with *Sonar*
 Build and publish a *Docker* image
 Deploy to *Heroku* (requires HEROKU_API_KEY set on CI service)
```

6. Let's provide a name to deploy; we can change it later if required:

```
? What CI/CD pipeline do you want to generate? Jenkins pipeline
? Would you like to perform the build in a Docker container ? No
? Would you like to send build status to GitLab ? No
? What tasks/integrations do you want to include ? Deploy to
*Heroku*
? *Heroku*: name of your Heroku Application ? store
```

Once the options are selected, JHipster will generate the files and log the following output on the console:

```
create Jenkinsfile
create src/main/docker/jenkins.yml
create src/main/resources/idea.gdsl
Congratulations, JHipster execution is complete!
```



If you want to use Travis instead of Jenkins, you can do so by choosing the Travis option and then publishing the repository to GitHub as a public repository. Once published, go to <https://github.com/<username>/<repoName>/settings/installations>, add Travis CI as a service, and follow the instructions. You can now see automated builds when you make commits. Refer to <https://docs.travis-ci.com/user/getting-started/> for details.

As you can see, we got `Jenkinsfile` generated at the root and a Docker image for Jenkins created in the `src/main/docker` directory. We also got an `idea.gdsl` file, which is used by IntelliJ IDEA for autocompletion.

## The Jenkinsfile and its stages

Let's take a look at the generated Jenkinsfile, which has our pipeline definitions, using the Groovy DSL:

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

node {
    stage('checkout') {
        checkout scm
    }
    stage('check java') {
        sh "java -version"
    }
    stage('clean') {
        sh "chmod +x gradlew"
        sh "./gradlew clean --no-daemon"
    }
    stage('nohttp') {
        sh "./gradlew checkstyleNohttp --no-daemon"
    }
    stage('npm install') {
        sh "./gradlew npm_install -PnodeInstall --no-daemon"
    }
    stage('backend tests') {
        try {
            sh "./gradlew test integrationTest -PnodeInstall
                --no-daemon"
        } catch(err) {
            throw err
        } finally {
            junit '**/build/**/TEST-*.xml'
        }
    }
    stage('frontend tests') {
        try {
            sh "./gradlew npm_run_test -PnodeInstall --no-daemon"
        } catch(err) {
            throw err
        } finally {
            junit '**/build/test-results/TESTS-*.xml'
        }
    }
    stage('packaging') {
        sh "./gradlew bootJar -x test -Pprod -PnodeInstall
            --no-daemon"
        archiveArtifacts artifacts: '**/build/libs/*.jar',
        fingerprint: true
    }
}
```

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

We have multiple stages defined running in a sequence, highlighted in bold; there are nine to be exact. It starts with a checkout of the branch from version control ending with deployment to Heroku (we will learn more about this in the following chapter).

The steps are quite straightforward as most of it is just triggering a Gradle task. Let's look at some of them:

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

The `checkout` stage does a local checkout of the source code revision that triggered the build:

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

This `check java` stage just prints the Java version installed on the Jenkins environment:

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

The `clean` stage first grants execution permission for the Gradle wrapper on a Unix-like OS and then executes the Gradle `clean` task. The `--no-daemon` flag disables the Gradle daemon feature, which is not required in a CI environment:

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

The `npm install` stage makes sure that Node.js and all of the NPM modules are installed by running `npm install` via Gradle.

The `-PnodeInstall` flag ensures that Node.js is installed first if not done already:

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

The `backend tests` stage runs all of the server-side integration and unit tests by triggering the Gradle test task. It will fail the Jenkins pipeline when there is an error and register the test reports on the Jenkins web UI using the JUnit plugin after the test run is complete:

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

Similar to the previous task, the `frontend tests` stage runs the client-side unit tests by triggering the NPM test command via a Gradle task. It will also fail the pipeline on an error and register the test reports on the Jenkins web UI:

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

The `packaging` stage triggers the Gradle `bootJar` task with the `prod` profile and archives the created JAR files with a unique fingerprint:

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

The final stage is for deployment and it uses a Gradle task for this. We will see this in detail in the following chapter. For now, let's comment out this stage. We will re-enable it later.

Now, let's commit everything to git by running these commands. Make sure you are on the master branch; if not, commit and merge the branch with the master:

```
> git add --all  
> git commit -am "add Jenkins pipeline for ci/cd"
```

Let's see how we can use the pipeline with the Jenkins server we set up earlier.

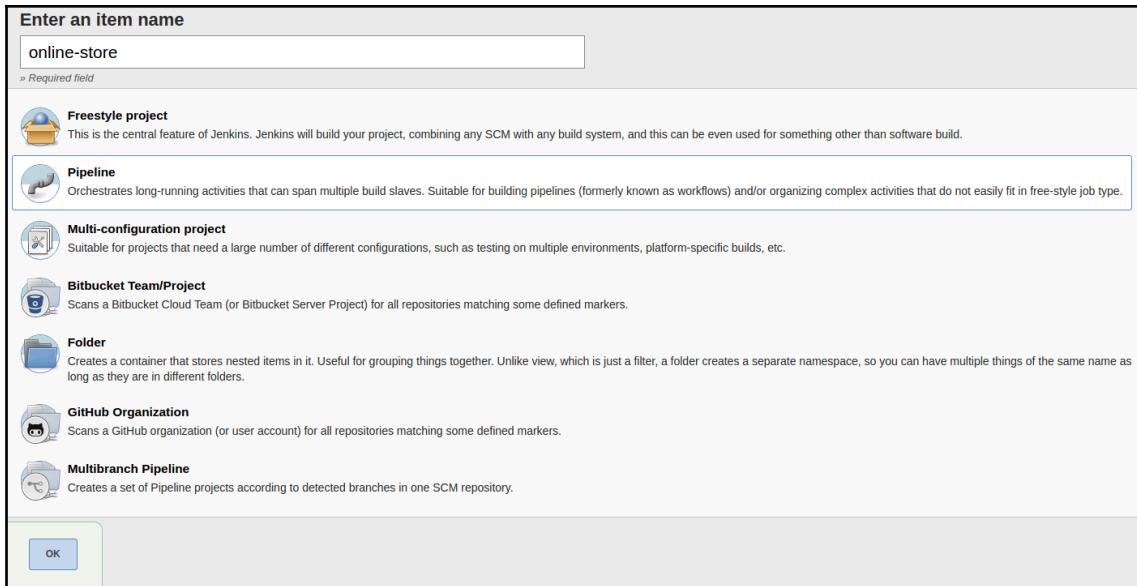
## Setting up the Jenkinsfile in a Jenkins server

Now that `Jenkinsfile` is ready, let's set up CI/CD for our application. First, we need to upload our application to a Git server, such as GitHub, GitLab, or Bitbucket. Let's use GitHub (<https://github.com/>) for this. Make sure you have an account created in GitHub first:

1. In GitHub, create a new repository (<https://github.com/new>); let's call it `online-store`. *Do not* check the **Initialize this repository with a README** option. Once created, you will see instructions to add code. Let's go with the option of **push an existing repository from the command line** by running the following commands inside our `online-store` application folder. Do not forget to replace `<username>` with your actual GitHub username:

```
> cd online-store  
> git remote add origin  
https://github.com/<username>/online-store.git  
> git push -u origin master
```

2. Now, go to the Jenkins server web UI by visiting `http://localhost:8989` and create a new job using the **create new jobs** link.
3. Enter a name, select **Pipeline** from the list, and click **OK**:



1. Scroll down or click on the **Build Triggers** section.
2. Select the **Poll SCM** checkbox.
3. Enter `H/01 * * * *` as the cron schedule value so that Jenkins polls our repository every minute and builds the code if there are new commits:

The screenshot shows the 'Build Triggers' configuration page for a Jenkins project. The tabs at the top are General, Build Triggers (selected), Advanced Project Options, and Pipeline. The 'Build Triggers' section contains the following configuration:

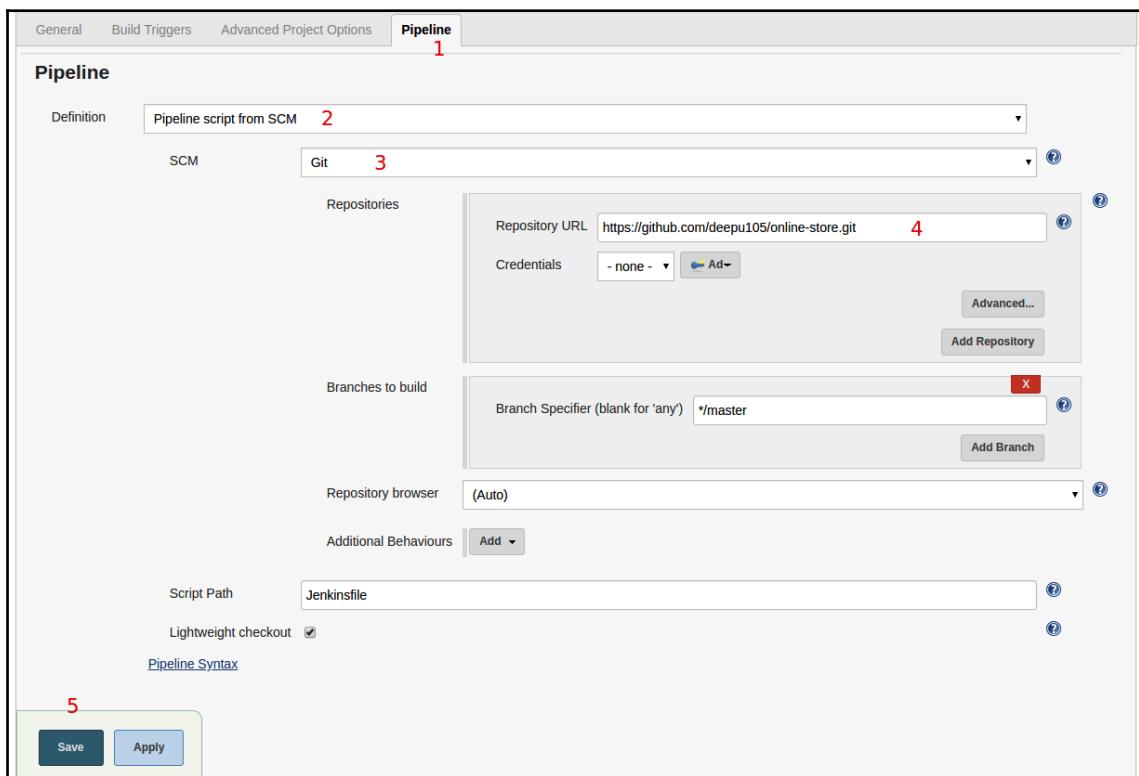
- 1** Build after other projects are built
- 2**  Poll SCM
- 3** Schedule: `H/01 * * * *`

Below the schedule, a note says 'Would last have run at Monday, 19 February, 2018 6:01:21 PM CET; would next run at Monday, 19 February, 2018 7:01:21 PM CET.'

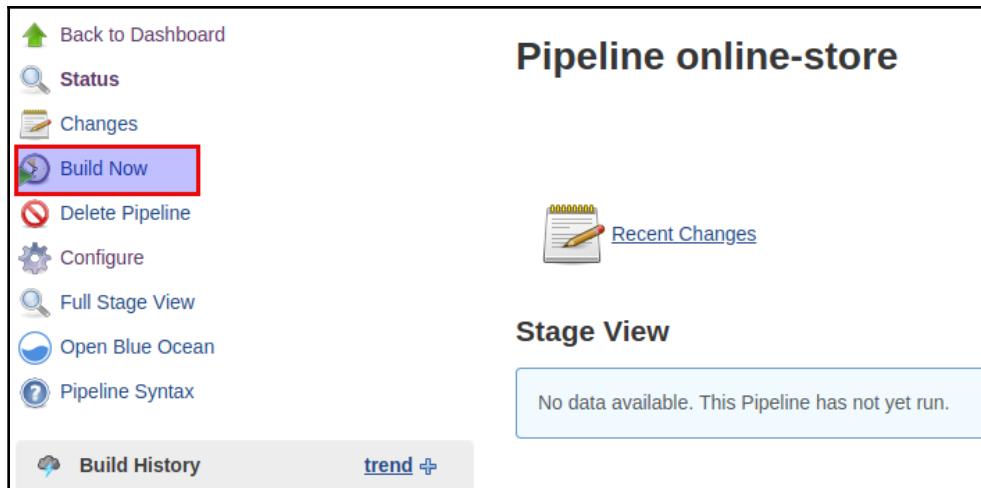
Other options in the 'Build Triggers' section include:

- Ignore post-commit hooks
- Disable this project
- Quiet period
- Trigger builds remotely (e.g., from scripts)

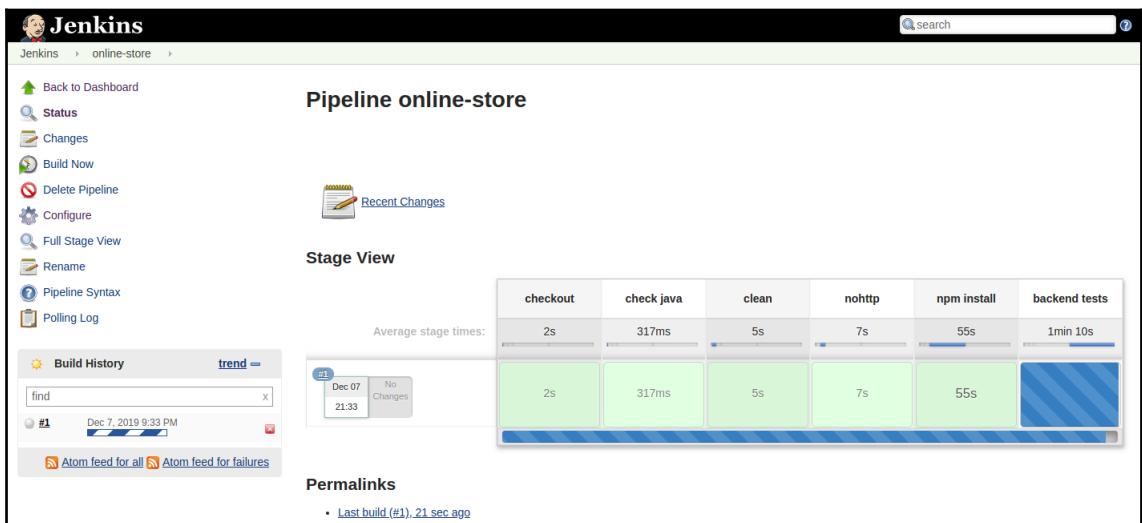
4. Next, do the following on the same page:
1. Scroll down or click on the **Pipeline** section.
  2. Select **Pipeline script from SCM** for the **Definition** field from the drop-down menu.
  3. Select **Git** for the **SCM** field from the drop-down menu.
  4. Add the **Repository URL** that we created in GitHub for the application.
  5. Finally, click **Save**:



5. Click on **Build Now** to trigger a new build to test our pipeline:



We should now see a build has started and its progress on the web UI, as shown in the following screenshot:



Congratulations! We have successfully set up CI/CD for our application. The builds will get triggered when you make new commits as well.

You can also view the pipeline status using the new UI from the Jenkins Blue Ocean plugin. Install the plugin from the Plugin Manager (click on **Jenkins** in the top menu and go to **Manage Jenkins** | **Manage Plugins** | **Available** and search for **Blue Ocean** and install it). The **Open Blue Ocean** link is available on the left-hand side menu. The builds will look as follows:

The screenshot shows the Jenkins Blue Ocean interface for the 'online-store' pipeline. At the top, there's a navigation bar with tabs for Pipelines, Administration, Logout, and links for Activity, Branches, and Pull Requests. Below the header, the pipeline name 'online-store' is displayed along with a cloud icon, a star icon, and a gear icon. A prominent 'Run' button is visible. The main content area shows a table of completed builds:

STATUS	RUN	COMMIT	MESSAGE	DURATION	COMPLETED
✓	2	–	polish 2 commits	4m 49s	15 days ago
✗	1	–	Started by user admin	4m 33s	18 days ago

Click on a build to view the pipeline. You can click on each stage of the progress indicator to list the steps from that stage, and then expand the list items to view the logs from that step:

The screenshot shows a detailed view of a Jenkins Blue Ocean pipeline for the 'online-store' project. The top navigation bar includes Pipeline, Changes, Tests, Artifacts, and other Jenkins icons. The pipeline details section shows the branch as '–', commit as '–', duration as '4m 49s' (started '15 days ago'), and notes that changes were made by d4uds. Below this is a timeline of stages: Start, checkout, check java, clean, install tools, backend tests, frontend tests, packaging, and End. Each stage has a green circular progress indicator with a checkmark. The 'packaging' stage is expanded, showing two log entries: './gradlew bootRepackage -x test -Pprod -PnodeInstall --no-daemon' (1m 35s) and 'Archive the artifacts' (1s). There are download icons next to the logs.

Congratulations. You have successfully set up a CI/CD pipeline using Jenkins.

## Summary

In this chapter, we looked at what CI/CD is and the tools supported by JHipster. We also learned how to set up Jenkins and created our CI/CD pipeline using JHipster and Jenkins. We also fixed our automated tests and made them run on the CI server.

In the next chapter, we will see how to deploy our application to *production* using a cloud-hosting provider such as Heroku.

# 7

## Going into Production

Our application is almost ready and it's time to go into production. Since this is the age of cloud computing, we will be deploying our application to a cloud provider—Heroku, to be specific. Before we go on and deploy our application into production, we need to make sure our application is production-ready in our local environment. It would also be beneficial to make ourselves familiar with technologies and tools that will be useful at this stage.

In this chapter, we will learn about the following:

- Introduction to Docker
- Starting the production database with Docker
- Introducing Spring profiles
- Packaging the application for local deployment
- Upgrading to the newest version of JHipster
- Introducing the deployment options supported by JHipster
- Production deployment to the Heroku Cloud

## Introduction to Docker

Docker is one of the most disruptive technologies to have taken center stage in the world of DevOps in recent times. Docker is a technology that enables operating system-level virtualization or containerization, and is also open source and free to use. Docker is intended for Linux, but it is possible to use it with Mac and Windows using tools such as Docker for Mac and Docker for Windows.

## Docker containers

When we talk about containers in the Docker world, we are technically talking about Linux containers. As stated by Red Hat on its website (<https://www.redhat.com/en/topics/containers/whats-a-linux-container>):

*"A Linux container is a set of processes that are isolated from the rest of the system, running from a distinct image that provides all files necessary to support the processes. By providing an image that contains all of an application's dependencies, it is portable and consistent as it moves from development to testing, and finally to production."*

Though the concept is not new, Docker makes it possible to create containers that are easy to build, deploy, version, and share. A Docker container only contains dependencies that are required for the application to run on the host OS; it shares the OS and other dependencies for the host system hardware. This makes a Docker container lighter than a **virtual machine (VM)** in terms of size and resource usage as it doesn't have to ship an entire OS and emulate virtual hardware. Hence, Docker made virtual machines obsolete in many of the traditional use cases that were handled using VM technologies. This also means that, with Docker, we will be able to run more applications on the same hardware compared to running with VMs. Docker containers are immutable instances of a docker image, which is a set of layers that describes the application that is being containerized. They contain the code, runtime, libraries, environment variables, and configuration files needed to run the application.

## Dockerfiles

A Dockerfile is a set of instructions that tells Docker how to build a Docker image. By running the `docker build` command on a specific Dockerfile, we will produce a Docker image that can be used to create Docker containers. Existing Docker images can be used as a base for new Dockerfiles, hence letting you reuse and extend existing images.

The following is what a Dockerfile of our application would look like:

```
FROM openjdk:8-jre-alpine

ENV SPRING_OUTPUT_ansi_enabled=ALWAYS \
    JHIPSTER_SLEEP=0 \
    JAVA_OPTS=""

CMD echo "The application will start in ${JHIPSTER_SLEEP}s..." && \
    sleep ${JHIPSTER_SLEEP} && \
    java ${JAVA_OPTS} -Djava.security.egd=file:/dev/.urandom -jar /app.war
```

```
EXPOSE 8080 5701/udp  
ADD *.war /app.war
```

The `FROM` instruction specifies the base image to use while initializing the build. Here, we specify OpenJDK 8 as our Java runtime.

The `ENV` instruction is used to set environment variables, and the `CMD` instruction is used to specify commands to be executed.

The `EXPOSE` instruction is used to specify the port that the container listens to during runtime.

Visit <https://docs.docker.com/engine/reference/builder/> for a complete reference.



Please note that we will not be using a Dockerfile directly, but instead we use JIB (<https://github.com/GoogleContainerTools/jib>), which is a tool that integrates with Gradle or Maven to build and push Docker images without needing a Docker daemon. This means you can build Docker images of our application in a computer that has JVM without needing Docker installed.

## Docker Hub

Docker Hub (<https://hub.docker.com/>) is the online registry provided by Docker. It can be used to publish public and private Docker images. This makes sharing and reusing Docker images extremely easy.

To get a Docker image from the registry, we just need to run `docker pull <image-name>`.

This makes it easy to use third-party tools without having to install them locally by just pulling and running the container from the registry. For example, if you want to run the Ubuntu OS via Docker, you can run the following command and it will provide you with a bash Terminal running on Ubuntu:

```
> docker run -e LANG=C.UTF-8 -e LC_ALL=C.UTF-8 -e TERM=$TERM -it --rm  
ubuntu bash
```

## Docker Compose

Docker Compose is a tool in the Docker platform that is used to define and run multi-container applications. It lets us define how a container will behave when it is run in production, and also lets us define other services that it depends on and how services work with each other. Each application is a service as it defines the behavior of the container, for example, what port it runs on, what environment variables it uses, and so on. A YAML file is used for this. A single `docker-compose.yml` file can define all the services that are required for a multi-container application and can then be started with a single command. We will learn more about Docker and Docker Compose in *Chapter 11, Deploying with Docker Compose*.



Visit <https://docs.docker.com/get-started/> to learn more about Docker.

The following table contains a list of useful commands for Docker and Docker Compose:

<code>docker build -t myapp:1.0.</code>	Build an image from the Dockerfile in the current directory and tag the image
<code>docker images</code>	List all images that are locally stored with the Docker engine
<code>docker pull alpine:3.4</code>	Pull an image from a registry
<code>docker push myrepo/myalpine:3.4</code>	Push an image to a registry
<code>docker login</code>	Log in to a registry (Docker Hub, by default)
<code>docker run --rm -it -p 5000:80 -v /dev/code alpine:3.4 /bin/sh</code>	Run a Docker container: --rm: Remove container automatically after it exits -it: Connect the container to the Terminal -p: Expose port 5000 externally and map to port 80 -v: Create a host-mapped volume inside the container alpine:3.4: The image from which the container is instantiated /bin/sh: The command to run inside the container
<code>docker stop myApp</code>	Stop a running container
<code>docker ps</code>	List the running containers
<code>docker rm -f \$(docker ps -aq)</code>	Delete all running and stopped containers
<code>docker exec -it web bash</code>	Create a new bash process inside the container and connect it to the Terminal

<code>docker logs --tail 100 web</code>	Print the last 100 lines of a container's logs
<code>docker-compose up</code>	Start the services defined in the <code>docker-compose.yml</code> file in the current folder
<code>docker-compose down</code>	Stop the services defined in the <code>docker-compose.yml</code> file in the current folder

In the next section, we'll learn how to run containers using Docker.

## Starting the production database with Docker

JHipster creates a Dockerfile for the application and provides `docker-compose` files for all the technologies we choose, such as the database, search engine, Jenkins, and so on, under `src/main/docker`:

```
└── app.yml - Main compose file for the application
└── hazelcast-management-center.yml - Compose file hazelcast management center
    ├── jenkins.yml - Compose file for Jenkins
    ├── monitoring.yml - Compose file for monitoring with Prometheus and Grafana
    ├── mysql.yml - Compose file for the database that we choose
    └── sonar.yml - Compose file for SonarQube
```

Let's look at how we can start our production database using Docker from the compose file provided under `src/main/docker/mysql.yml`. You will need to use a Terminal for the following instructions:

1. Run `docker --version` and `docker-compose --version` to ensure these are installed.
2. Run `docker ps` to list the running containers. If you are not running any containers, you should see an empty list.
3. Let's start the DB by running `docker-compose -f src/main/docker/mysql.yml up`.

You will see the following console output:

```
$ docker-compose -f src/main/docker/mysql.yml up
Recreating docker_store-mysql_1 ...
Recreating docker_store-mysql_1 ... done
Attaching to docker_store-mysql_1
store-mysql_1 | 2018-03-03T14:10:42.518245Z 0 [Note] mysqld (mysqld 5.7.20) starting as process 1 ...
store-mysql_1 | 2018-03-03T14:10:42.521178Z 0 [Note] InnoDB: PUNCH HOLE support available
store-mysql_1 | 2018-03-03T14:10:42.521196Z 0 [Note] InnoDB: Mutexes and rw_locks use GCC atomic builtins
store-mysql_1 | 2018-03-03T14:10:42.521208Z 0 [Note] InnoDB: Uses event mutexes
store-mysql_1 | 2018-03-03T14:10:42.521204Z 0 [Note] InnoDB: GCC builtin __atomic_thread_fence() is used for memory barrier
store-mysql_1 | 2018-03-03T14:10:42.521209Z 0 [Note] InnoDB: Compressed tables use zlib 1.2.3
store-mysql_1 | 2018-03-03T14:10:42.521213Z 0 [Note] InnoDB: Using Linux native AIO
store-mysql_1 | 2018-03-03T14:10:42.521832Z 0 [Note] InnoDB: Number of pools: 1
store-mysql_1 | 2018-03-03T14:10:42.522727Z 0 [Note] InnoDB: Using CPU crc32 instructions
store-mysql_1 | 2018-03-03T14:10:42.524890Z 0 [Note] InnoDB: Initializing buffer pool, total size = 128M, instances = 1, chunk size = 128M
store-mysql_1 | 2018-03-03T14:10:42.531726Z 0 [Note] InnoDB: Completed initialization of buffer pool
store-mysql_1 | 2018-03-03T14:10:42.533641Z 0 [Note] InnoDB: If the mysqld execution user is authorized, page cleaner thread priority can be changed.
store-mysql_1 | 2018-03-03T14:10:42.548765Z 0 [Note] InnoDB: Highest supported file format is Barracuda.
store-mysql_1 | 2018-03-03T14:10:42.550433Z 0 [Note] InnoDB: Log scan progressed past the checkpoint lsn 12213183
store-mysql_1 | 2018-03-03T14:10:42.550454Z 0 [Note] InnoDB: Doing recovery: scanned up to log sequence number 12213192
store-mysql_1 | 2018-03-03T14:10:42.550458Z 0 [Note] InnoDB: Database was not shutdown normally!
store-mysql_1 | 2018-03-03T14:10:42.550461Z 0 [Note] InnoDB: Starting crash recovery.
```



If you want to run the service in the background, pass the `-d` flag to the command. `docker-compose -f src/main/docker/mysql.yml up -d` will let you continue using the same Terminal without having to switch to another.

Now, if you run `docker ps` again, it should list the database service that we started:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
f16076e9f661        mysql:5.7.20      "docker-entrypoint..."   6 minutes ago    Up 4 seconds          0.0.0.0:3306->3306/tcp   docker_store-mysql_1
```

Before we prepare our application for production, let's talk a little bit about Spring profiles.

## Introducing Spring profiles

Spring profiles (<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-definition-profiles-java>) let you change the way your application behaves based on environments. This is achieved using the `@Profile` annotations and profile-specific configuration files, which can be activated by specifying the `spring.profiles.active` property. Based on the profile that we set here, Spring will choose the appropriate `application.properties` or `application.yml` files and will include/exclude components that are included/excluded for the specific profile using the `@Profile` annotation in the Java source code.

For example, if we set `spring.profiles.active=prod`, all the Spring components that have `@Profile("prod")` will be instantiated and any component that has `@Profile("!prod")` will be excluded. Similarly, Spring will load and use the `application-prod.yml` or `application-prod.properties` file if it is available on the classpath.

JHipster configures a `dev` and `prod` profile by default, and includes the `application-dev.yml` and `application-prod.yml` files in the `src/main/resources/config` folder, along with the base `application.yml` file. JHipster goes a step further and provides a `dev` and a `prod` profile for the Gradle build as well (also available for Maven) so that we can build/run the application for a particular profile, which is very handy. Here are the profile and database configurations defined in the `application-dev.yml` file:

```
...
spring:
  profiles:
    active: dev
    include:
      - swagger
      # Uncomment to activate TLS for the dev profile
      #- tls
...
datasource:
  type: com.zaxxer.hikari.HikariDataSource
  url: jdbc:h2:file:./build/h2db/db/store;DB_CLOSE_DELAY=-1
  username: store
  password:
  hikari:
    poolName: Hikari
    auto-commit: false
...
...
```

The following profiles are available in a JHipster application:

dev	Tuned for development and productivity, it enables Spring dev tools, in-memory databases, and so on
prod	Tuned for production, it focuses on performance and stability
swagger	Enables Swagger documentation for the API
no-liquibase	Disables Liquibase and is useful in production environments where you don't want Liquibase to run
tls	Runs the application in TLS mode for security

Now, let's look at how we can package our application with the production profile.

# Packaging the application for local deployment

Now, let's build our application and deploy it locally. This can be done in two ways, either using Docker or by building and executing a WAR file.

## Building and deploying using Docker

JHipster uses JIB (<https://github.com/GoogleContainerTools/jib>) to build Docker images. JIB can build and push Docker images without the need of a Docker daemon and, hence, is an ideal choice for CI environments as well. The JIB plugin is integrated with our Gradle build. The configuration that's used is defined in `gradle/docker.gradle` and is as follows:

```
jib {
    from {
        image = "adoptopenjdk:11-jre-hotspot"
    }
    to {
        image = "store:latest"
    }
    container {
        entrypoint = ["bash", "-c", "chmod +x /entrypoint.sh && sync &&
            /entrypoint.sh"]
        ports = ["8080", "5701/udp" ]
        environment = [
            SPRING_OUTPUT_ANSI_ENABLED: "ALWAYS",
            JHIPSTER_SLEEP: "0"
        ]
        creationTime = "USE_CURRENT_TIMESTAMP"
    }
}
```

This configuration is quite straightforward. The base image that's used is specified, which is `adoptopenjdk:11-jre-hotspot` in this case. The output image name is specified, followed by container-related options such as entry point scripts, exposed ports, and environment variables.

Let's use the JIB Gradle task to build our Docker image:

Use the `./gradlew tasks` command to list all available tasks.



1. In your Terminal, go to the project root folder and execute `./gradlew clean bootJar -Pprod jibDockerBuild`:
  - `bootJar`: This builds an executable archive (JAR) file for the application.
  - `-Pprod`: This specifies the profile to use.
  - `jibDockerBuild`: This builds a Docker image based on the configuration present in the `gradle/docker.gradle` file.



If you want to build a WAR file instead of a JAR file, use `./gradlew -Pprod -Pwar clean bootWar` instead of the previous command.

2. Once the task has completed successfully, we can deploy our app by running the following command:

```
> docker-compose -f src/main/docker/app.yml up
```

This will also start the MySQL DB, if you haven't started it already. If you already have it running from the previous step, then `docker-compose` will just skip it.

Our application will be ready once we see the following output in the console. As you can see, it's running with the `prod` and `swagger` profiles:

```
store-app_1 | -----
store-app_1 | Application 'store' is running! Access URLs:
store-app_1 |   Local:          http://localhost:8080
store-app_1 |   External:       http://172.18.0.2:8080
store-app_1 |   Profile(s):    [prod, swagger]
store-app_1 | -----
```

Visit `http://localhost:8080` in your favorite browser to see the application in action.

## Building and deploying an executable archive

If you prefer not to use Docker, then you could deploy the app with a production profile locally by completing the following steps:

1. First, make sure that MySQL DB is running from the previous step; otherwise, start it using `docker-compose -f src/main/docker/mysql.yml up -d`.
2. Now, let's create an executable archive for the prod profile by running `./gradlew clean bootJar -Pprod`.
3. Once the build is successful, there will be an archive (JAR) created under `build/libs`. The `store-0.0.1-SNAPSHOT.jar` file is an executable archive that can be run directly on a JVM.
4. Let's use the executable archive. Just run `java -jar build/libs/store-0.0.1-SNAPSHOT.jar`.

Once the application starts up, you will see the URL printed on the console.

Visit `http://localhost:8080` in your favorite browser to see the application in action.

## Upgrading to the newest version of JHipster

JHipster provides an "upgrade" sub-generator (<http://www.jhipster.tech/upgrading-an-application/>) to help you upgrade an application with a new JHipster version. It is quite useful as it automates a lot of manual steps for you and the only thing you need to do is resolve merge conflicts, if there are any, after the upgrade is complete. Let's upgrade our application.

In your Terminal, execute the `jhipster upgrade` command. The upgrade process will start if there is a new version of JHipster available; otherwise, the process will exit with an error message.

Once the process starts, you will see a detailed console log of what is going on. As you can see, this sub-generator uses the global JHipster version instead of the local one, unlike other sub-generators:

```
INFO! Using JHipster version installed globally
INFO! Executing jhipster:upgrade
INFO! Options: from-cli: true
Welcome to the JHipster Upgrade Sub-Generator
This will upgrade your current application codebase to the latest JHipster
version
✓ Checking for new blueprint versions
✓ Done checking for new version of blueprints
Looking for latest generator-jhipster version...
Looking for latest generator-jhipster version...
6.3.1
✓ New generator-jhipster version found: 6.3.1
  info git rev-parse -q --is-inside-work-tree
true
✓ Git repository detected
...
Switched to a new branch 'jhipster_upgrade'
✓ Created branch jhipster_upgrade
...
✓ Cleaned up project directory

Installing generator-jhipster 6.2.0 locally
...
+ generator-jhipster@6.2.0
added 417 packages from 159 contributors and audited 2754226 packages in
39.961s
found 9 vulnerabilities (3 moderate, 6 high)
  run `npm audit fix` to fix them, or `npm audit` for details
✓ Installed generator-jhipster@6.2.0
...
✓ Committed with message "Generated with JHipster 6.2.0"
  info git checkout -q master
✓ Checked out branch "master"
  info git --version
git version 2.21.0
  info git merge --strategy=ours -q --no-edit --allow-unrelated-
histories jhipster_upgrade
✓ Current code has been generated with version 6.2.0
  info git checkout -q jhipster_upgrade
✓ Checked out branch "jhipster_upgrade"
...
✓ Installed generator-jhipster@6.3.1
...
```

```
✓ Successfully regenerated application with JHipster 6.3.1 and
  info Removing src/main/resources/config/tls/keystore.p12
  info git add -A
warning: LF will be replaced by CRLF in gradlew.bat.
The file will have its original line endings in your working directory
  info git commit -q -m "Generated with JHipster 6.3.1 and " -a --allow-
empty --no-verify
✓ Committed with message "Generated with JHipster 6.3.1 and "
  info git checkout -q master
✓ Checked out branch "master"
...
✓ Merge done!
...
✓ Upgraded successfully.
...
INFO! Congratulations, JHipster execution is complete!
```

The sub-generator does the following, in order:

1. Checks whether there is a new version of JHipster available (not applicable if you are using `--force`).
2. Checks whether the application is already initialized as a Git repository; otherwise, JHipster will initialize one for you and commit the current code base to the master branch.
3. Checks to ensure that there are no uncommitted local changes in the repository. The process will exit if it finds any uncommitted changes.
4. Checks whether a `jhipster_upgrade` branch exists. If not, a branch is created.
5. Checks out the `jhipster_upgrade` branch.
6. Upgrades JHipster to the latest available version globally.
7. Cleans the current project directory.
8. Regenerates the application using the `jhipster --force --with-entities` command.
9. Commits the generated code to the `jhipster_upgrade` branch.
10. Merges the `jhipster_upgrade` branch back to the original branch from where the `jhipster upgrade` command was launched.

Let's see what has changed before we resolve the merge conflicts. Check the changes that have been staged. Carefully check the changes to make sure everything is in order, especially in the files where we made customizations earlier. My changelog looks like this; note that I truncated the bottom as there were 147 updated files:

MERGE CHANGES	
↳ package.json	C
↳ gradle-wrapper.properties	C
↳ CustomerResourceIntTest.java	C
STAGED CHANGES	
{.} .yo-rc.json	M
ⓘ README.md	M
↳ gradle.properties	M
{.} tsconfig.json	M
{.} tsconfig-aot.json	M
{.} tslint.json	M
➡ TokenProvider.java	M
➡ app-routing.module.ts	M
➡ app.constants.ts	M
➡ app.module.ts	M
➡ activate.service.ts	M
➡ password-strength-bar.component.ts	M
➡ password.service.ts	M
➡ password-reset-finish.service.ts	M
➡ password-reset-init.service.ts	M
➡ register.component.ts	M
➡ register.service.ts	M
➡ audits.component.ts	M
➡ audits.service.ts	M
➡ configuration.service.ts	M
➡ health.service.ts	M
➡ logs.component.ts	M
➡ logs.service.ts	M

Thankfully, I only have three conflicts, so they should be easy to resolve. The conflict in package.json arises from the change we made to integrate Bootswatch. Carefully resolve the conflict stage in the file and move on to the next file.



The result of the upgrade process depends on when the command is run, the version that was used to generate the application, and the latest JHipster release version available. Hence, the number of files that have changed and conflicts arising will vary greatly, depending on these variables. The conflict resolution needs to be carefully done while keeping this in mind. What I have shown here is just a sample.

Once all the conflicts have been resolved, stage the files and commit them, as shown here:

```
> git add --all  
> git commit -am "update to latest JHipster version"
```

Ensure that everything works. Run the server-side and client-side tests using `./gradlew test integrationTest npm_test` and start the application to verify this by running the `./gradlew clean bootRun` command.

In the next section, we'll look at what deployment options are available for us.

## Deployment options supported by JHipster

Now that we have verified our production build by deploying it locally, let's see how we can take it to actual production by using a cloud service. JHipster supports most of the cloud platforms out of the box and provides special sub-generator commands for popular ones such as Heroku, Azure, Google App Engine, Cloud Foundry, and AWS.

JHipster also supports platforms such as OpenShift and Kubernetes, but we will look at them in upcoming chapters as they are more geared toward microservices. In theory, though, you could use them for Monolith deployments as well.

## Heroku

Heroku (<https://www.heroku.com/>) is the cloud platform from Salesforce. It lets you deploy, manage, and monitor your applications on the cloud. Heroku has a focus on applications rather than on containers, and supports languages ranging from Node.js, to Java, to Golang. JHipster provides the Heroku sub-generator, which was built and is maintained by Heroku, making it easy to deploy JHipster apps to the Heroku Cloud. It makes use of the Heroku CLI and you need a Heroku account to use it. The sub-generator can be used to deploy and update your application to Heroku.

Visit <http://www.jhipster.tech/heroku/> for more information.

## Cloud Foundry

Cloud Foundry is a multi-cloud computing platform governed by the Cloud Foundry Foundation. It was originally created by VMWare and is now under Pivotal, the company behind the Spring Framework. It offers a multi-cloud solution that is currently supported by **Pivotal Cloud Foundry (PCF)**, **Pivotal Web Services (PWS)**, Atos Canary, SAP Cloud Platform, and IBM Bluemix, among others. The platform is open source, so it can be used to set up your own private instance. JHipster provides a sub-generator so that you can deploy JHipster applications to any Cloud Foundry provider easily. It makes use of the Cloud Foundry command-line tool.

Visit <http://www.jhipster.tech/cloudfoundry/> for more information.

## Amazon Web Services

**Amazon Web Services (AWS)** is the leading cloud-computing platform that offers Platform, Software, and Infrastructure as a Service. AWS offers Elastic Beanstalk as a simple platform that you can use to deploy and manage your applications on the cloud. JHipster provides a sub-generator so that you can deploy JHipster applications to AWS or Boxfuse (<http://www.jhipster.tech/boxfuse/>), an alternative service.

Visit <http://www.jhipster.tech/aws/> for more information.

## Google App Engine

**Google App Engine (GAE)** is a fully managed serverless application platform from Google Cloud Platform that can be used for deploying simple applications. The platform supports various languages, including Java. JHipster provides a sub-generator, which is built and maintained by the Google Cloud team, to deploy to the App Engine easily. It makes use of the `gcloud` CLI and you need a Google Cloud account to use it.

Visit <https://cloud.google.com/appengine/> for more information.

## Azure Spring Cloud

Azure Spring Cloud (<https://azure.microsoft.com/en-us/services/spring-cloud/>) is a fully managed platform from Microsoft Azure tailored for Spring Boot applications. It is jointly built and managed by Microsoft and Pivotal, the company behind the Spring Framework.

As a result, it is well-integrated into the Azure Cloud, with out-of-the-box security and monitoring from Azure, and it also benefits from Pivotal's cloud-native solutions such as the Spring Cloud Config Server and the Spring Cloud Discovery Server (based on Netflix Eureka).

The platform is particularly well suited to run Spring Boot microservices as it provides all the necessary services, such as configuration, service discovery, scalability, and monitoring. It can also run monoliths, although Azure App Service (see next section) might be a better option for those.

As JHipster generates standard Spring Boot applications, its monoliths, microservices, and gateways all run on Azure Spring Cloud. As both the Spring Cloud Discovery Server (provided and managed by Azure Spring Cloud) and the JHipster Registry (provided, by default, by JHipster) are built on top of Netflix Eureka, it is possible to swap both solutions: you can generate a microservice using the JHipster Registry and run it on Azure Spring Cloud without any code change.

In order to ease deployment on Azure Spring Cloud, JHipster provides a dedicated sub-generator. It makes use of the Azure CLI and will require that you have an Azure account in order to create a Spring Cloud cluster and deploy your application.

Visit <https://www.jhipster.tech/azure/> for more information on this sub-generator.

## Azure App Service

Azure App Service (<https://azure.microsoft.com/en-us/services/app-service/>) is a **Platform as a Service (PaaS)** provided by Microsoft Azure. It supports a wide variety of languages and frameworks, including Java.

This is a more generic solution than Azure Spring Cloud (see the previous section), but it is also more mature and less expensive (as it does not provide Spring Cloud services such as the configuration server or the discovery server). As a result, it is a good option for deploying Spring Boot monoliths, including those generated with JHipster.

There are two ways to deploy a JHipster application on Azure App Service:

- Bundle the application as a Docker image. As JHipster has great Docker support, it is easy to publish that image and use it on Azure App Service.
- Provide an executable JAR file and run it on Azure App Service. The second option is the one we recommend since it is faster and easier to build an executable JAR file, and also because that will let Azure manage the operating system, Java versions, and upgrades, resulting in lower security risks.

As with Azure Spring Cloud, JHipster provides a sub-generator so that you can deploy your application to Azure App Service as well. You will also need the Azure CLI and an Azure account. At the moment, it also requires the use of Maven as it uses an Azure Maven plugin to do this deployment, but please note that this might change in the future (Azure is working on a Gradle plugin, and the JHipster team is working to deploy using only the Azure CLI and no plugin).

Visit <https://www.jhipster.tech/azure/> for more information on this sub-generator.

## Production deployment to the Heroku Cloud

We need to choose a cloud provider. For this demo, we'll choose Heroku.



Though the Heroku account is free and you get free credits, you will have to provide your credit card information to use MySQL and other add-ons. You will only be charged if you exceed the free quota.

Let's deploy our application to Heroku by completing the following steps:

1. First, you need to create an account on Heroku (<https://signup.heroku.com/>). It is free and you get free credits as well.
2. Install the Heroku CLI tool by following the instructions at <https://devcenter.heroku.com/articles/heroku-cli>.
3. Verify that the Heroku CLI is installed by running `heroku --version`.
4. Log in to Heroku by running `heroku login`. When prompted, enter your Heroku email and password.
5. Now, run the `jhipster heroku` command. You will start seeing questions.
6. Choose a name you like when asked **Name to deploy as: (store)**. By default, it will use the application name. Try to choose a unique name since the Heroku namespace is shared.
7. Next, you will be asked to choose a region—? **On which region do you want to deploy?** Choose between the US and EU, and proceed.
8. Next, you will be asked to choose a deployment type—? **Which type of deployment do you want?** Choose JAR, and proceed.
9. The generator will create the required files and you need to accept changes on the Gradle build files.

The console output will look like this:

```
> jhipster heroku
INFO! Using JHipster version installed locally in current project's node_modules
INFO! Executing jhipster:heroku
INFO! Options: from-cli: true
Heroku configuration is starting
? Name to deploy as: jhbook-online-store
? On which region do you want to deploy ? eu
? Which type of deployment do you want ? JAR (compile locally)

Initializing Git repository
Initialized empty Git repository in /home/deepu/Documents/jhipster-book/v2/e-commerce-app/online-store/.git/

Installing Heroku CLI deployment plugin

Creating Heroku application and setting up node environment
https://jhbook-online-store.herokuapp.com/ | https://git.heroku.com/jhbook-online-store.git

Provisioning addons
Created Database addon

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

Building application
> Task :bootBuildInfo
```

The generated .yml files add Heroku-specific configurations for the application. The Procfile contains the specific command that will be executed on Heroku for the application. The Gradle build is also modified to include dependencies required by Heroku.

After generating the files, it will build the application and start uploading artifacts. This may take several minutes based on your network latency. Once this has been successfully completed, you should see the following screen:

```
Deploying application

Uploading your application code.
This may take several minutes depending on your connection speed...
Uploading store-0.0.1-SNAPSHOT.jar
----> Packaging application...
    - app: jhbook-online-store
    - including: build/libs/store-0.0.1-SNAPSHOT.jar
----> Creating build...
    - file: slug.tgz
    - size: 64MB
----> Uploading build...
    - success
----> Deploying...
remote:
remote: ----> heroku-deploy app detected
remote: ----> Installing JDK 1.8... done
remote: ----> Discovering process types
remote:       Procfile declares types -> web
remote:
remote: ----> Compressing...
remote:       Done: 115M
remote: ----> Launching...
remote:       Released v6
remote:       https://jhbook-online-store.herokuapp.com/ deployed to Heroku
remote:
----> Done

Your app should now be live. To view it run
      heroku open
And you can view the logs with this command
      heroku logs --tail
After application modification, redeploy it with
      jhipster heroku
INFO! Congratulations, JHipster execution is complete!
```

Now, run the `heroku open -a jhbook-online-store` command to open the deployed application in a browser. That's it—you have successfully deployed your application to Heroku with a few commands.

When you update the application further, you can run the `jhipster heroku` command again to redeploy, or you can rebuild the package using `./gradlew -Pprod bootJar` and then redeploy it using the `heroku deploy:jar --jar build/libs/*.jar` command.

Don't forget to commit the changes that you've made to `git` by executing the following command:

```
> git add --all  
> git commit -am "add heroku configuration"
```

That is it—we have created and deployed a monolithic, full stack application to the cloud.

## Summary

Deployment to production is one of the most important phases of application development and is the most crucial one as well. With the help of JHipster, we deployed our application to a cloud provider with ease. We also learned about Docker and the various other deployment options available. We also made use of the upgrade sub-generator to keep our application up to date with JHipster.

So far, we've seen how we can develop and deploy a monolithic e-commerce application using JHipster. We started with a monolith and, in the upcoming chapters, we will see how we can scale our application into a microservice architecture with the help of JHipster. In the next chapter, we will learn about different microservice technologies and tools. So, stay tuned!

# 4

## Section 4: Converting Monoliths to Microservice Architecture

In this section, you will first be introduced to the different options available in the JHipster microservice stack. Moving on, you will see how a JHipster monolith web application is converted into a full-fledged microservice architecture with a gateway, registry, monitoring console, and multiple microservices. You will also examine the generated code and components, such as the JHipster Registry, JHipster Console, API gateways, and JWTs.

In the last chapter of this section, you will see how to run the generated applications in a local environment.

This section comprises the following chapters:

- Chapter 8, *Introduction to Microservice Server-Side Technologies*
- Chapter 9, *Building Microservices with JHipster*
- Chapter 10, *Working with Microservices*

# 8

# Microservice Server-Side Technologies

Wasn't it easy to develop a production-ready monolithic application with JHipster? So far, we have created an application from scratch, added a few entities with JDL Studio, and then deployed it to the production environment along with tests. We have also added a continuous integration and continuous delivery pipeline. Wasn't the experience faster, easier, and better than coding everything from scratch?

So what's next? Yes, you guessed it right – **microservices!**

Microservices is the buzzword everywhere these days. Many companies out there are trying to solve their problems with microservices. We already saw an overview of the benefits of microservices in Chapter 1, *Introduction to Modern Web Application Development*.

In this chapter, we will learn about the following:

- Benefits of microservices over monoliths
- Components that we need to build a complete microservices architecture

After that, we will see how easy it is to create a microservices architecture using the options JHipster provides.

# Microservice applications versus monoliths

The benefits of microservices architectures can be better understood by comparing them with monolithic architectures.



The benefits of microservices over monoliths are phenomenal when they are designed and deployed correctly for the appropriate use case.

Microservices are not a silver bullet; in many use cases, they might actually bring more issues than benefits. So choosing microservices should be done with great care and based on use cases.

It is not as simple as splitting a monolithic application based on structure, component, or functionality and then deploying them as individual services. This will not work out.

Converting a monolithic application or even a monolithic design into microservices needs a clear vision of the product. It includes knowledge of what part of the project will change and what part will be consistent. We must have low-level details, such as which entities we should group together and those that can be separated.

This clearly illustrates the need for an ever-evolving model. It is much easier to split the technologies used in the application, but not the interdependent models or the business logic of the application. So it is essential to place the project's primary focus on the core domain and its logic.

Microservices should be independent. They will fail when one component is tightly coupled with another. The trickiest part is identifying and segregating the components.

When we have done that, it offers the following benefits over monolithic applications.

## Scalability

The monolithic code is a single unit. Thus, all parts of the application share the same memory. For a bigger system, we need to have a bigger infrastructure. When the application grows, we need to scale the infrastructure as needed. The scaling of an already bigger infrastructure is always a difficult and costlier task for operations.

Even though they have all the necessary code to handle anything in the product at a single place (no need to worry about latency or availability), it is difficult to handle the resources that it consumes to run and it is definitely not scalable. If one part of the application fails, then the whole product will be impacted. When one thread or query of the product clings on to the memory, then the impact will be seen by millions of our customers.

Microservices, on the other hand, require less memory per service to run since we are splitting the application into smaller components, which, in turn, could reduce the infrastructure's cost for high-bandwidth applications. For example, it is cheaper to run ten 2 GB instances (costs ~\$170 per month on AWS) than running a single 16 GB instance (costs ~\$570 per month on AWS). Each component runs in its own environment, which makes microservices much more developer-friendly (if you are a large team) and cloud-native. Similarly, microservices also increase the throughput across services. A memory-intensive operation on one service will not affect any other service as the load is distributed.

## Efficiency

Monolithic architecture, over a period of time, will remove the agility of a team, which will delay the application rollout. This means people will tend to invest more time in finding a workaround to fix a problem when a new feature is added or something in the existing feature breaks. The monolithic architecture will bring a greater amount of inefficiency for larger projects, which in turn, increases the technical debt.

Microservices, on the other hand, reduce the technical debt in terms of architecture since everything is reduced to individual components. Teams tend to be more agile and they will find handling changes easier.



The less code there is, the fewer bugs there are, meaning less pain and a shorter time to fix.

## Time constraint

Monolithic applications are more time consuming to work with. Imagine there is a big monolithic application and you have to reverse an `if condition` in your service layer. After changing the code, it has to be built, which usually takes a few minutes, and then you must test the entire application, which will reduce the team's performance.

You can reboot or reload an application in seconds for a microservices architecture. When you have to reverse an `if condition`, you need not wait for minutes to build and deploy the application to test; you can do it in seconds. This will decrease the time it takes to do mundane tasks.



Faster iterations/releases and decreased downtime are the key things to increase user engagement and user retention, which, in turn, results in better revenue.

A human mind (unless you are superhuman) can handle only a limited amount of information. So cognitively, microservices help people to reduce the clutter and focus on the functionality. This enables better productivity and faster rollouts.

To sum up, embracing microservices for an appropriate use case will do the following:

- Maximize productivity
- Improve agility
- Improve the customer experience
- Speed up development/unit testing (if designed properly)
- Improve revenue

## Building blocks of a microservices architecture

Running a microservices architecture requires a lot of components/features and involves a lot of advanced concepts. For the sake of understanding these concepts, imagine we have a microservice-based application for our e-commerce shopping website. This includes the following services:

- **Pricing services:** Responsible for giving us the price of the product based on demand
- **Demand services:** Responsible for calculating the demand for the product based on sales and remaining stock
- **Inventory services:** Responsible for tracking the quantity left in the inventory, and many other services

Some of the concepts we will see in this section are the following:

- Service registry
- Service discovery
- Health check

- Dynamic routing and resiliency
- Security (authentication and authorization)
- Fault tolerance and failover

## Service registry

Microservices are independent, but many use cases will need them to be interdependent. This means that for some services to work properly, they need data from another service, which, in turn, may or may not depend on other services or sources.

For example, our pricing service will directly depend on the demand service, which, in turn, depends on the inventory service. But these three services are completely independent, that is, they can be deployed on any host, port, or location and scaled at will.

If the pricing service wants to communicate with the demand service, it has to know the exact location to which it can send requests to get the required information. Similarly, the demand service should know about the inventory service's details in order to communicate. We will not know the details of the location beforehand as the services can be scaled independently.

So we need a service registry that registers all the services and their locations and configuration. All services should register themselves with this registry service when the service is started and deregister themselves when the service goes down.



The service registry should act as a database of services, recording all the available instances and their details.

## Service discovery

The service registry has details of the services available and acts as the database. But in order to find out where the required service is and which services to connect, we need to have service discovery. So, service discovery is the end-to-end process and the service registry is the data store used by that process. They both work hand in hand.

When the pricing service wants to communicate with the demand service, it needs to know the network location of the demand service. In the case of traditional architecture, this is a fixed physical address, but in the microservices world, this is a dynamic address that is assigned and updated dynamically.

The pricing service (client) will have to locate the demand service in the service registry and determine the location and then load balance the request of the available demand service. The demand service, in turn, will respond to the request of the requested client (pricing service).

Service discovery is used to discover the exact service to which the client should connect to, in order to get the necessary details.



Service discovery helps the API gateway to discover the right endpoint for a request.

They will also have a load balancer, which regulates the traffic and ensures the high availability of the services.

Based on the location where load balancing happens, the service discovery is classified into:

- A client-side discovery pattern
- A server-side discovery pattern

## Client-side discovery pattern

Load balancing will happen on the client service side. The client service will determine where to send the request and the logic of load balancing will be in the client service. For example, Netflix Eureka (<https://github.com/Netflix/eureka>) is a service registry. It provides endpoints to register and discover the services.

When the pricing service wants to invoke the demand service, it will connect to the service registry and then find the available services. Then, based on the load balancing logic configured, the pricing service (client) will determine which demand service to request.

The services will then do an intelligent and application-specific load balancing. On the downside, this adds an extra layer of load balancing to every service, which is an overhead.

## Server-side discovery pattern

The pricing service will request the load balancer to connect to the demand service. Then, the load balancer will connect to the service registry to determine the available instance, and then route the request based on the load balancing configured.

For example, in Kubernetes, each pod will have its own server or proxy. All the requests are sent through this proxy (which has a dedicated IP and port associated with it).

The load balancing logic is moved away from the service and isolated into a separate service. On the downside, it requires yet another highly available service to handle the requests.

## Health check

In the microservices world, instances can start, change, update, and stop at random. They can also scale up and down based on their traffic and other settings. This requires a health check service that will constantly monitor the availability of the services.

Services can send their status periodically to this health check service, and this keeps a track of the health of the services. When a service goes down, the health check service will stop getting the heartbeat from the service. Then, the health check service will mark the service down and cascade the information to the service registry. Similarly, when the service resumes, the heartbeat is sent to the health check service. Upon receiving a few positive heartbeats, the service is marked **UP** and then the information is sent to the service registry.

The health check service can check for health in two ways:

- **Push configuration:** All the services will send their heartbeat periodically to the health check service.
- **Pull configuration:** A single health check service instance will query for the availability of the systems periodically.

This also requires a **high availability system**. All the services should connect to this service to share their heartbeat and this has to connect to the service registry to tell them whether a service is available. This normally is part of the service discovery system.

## Dynamic routing and resiliency

The health check services will track the health of available services and send details to the service registry regarding the health of services.



Based on the health of the services, the service discovery process should intelligently route requests to healthy instances and shut down the traffic to unhealthy instances.

Since the services dynamically change their location (address/port) every time a client wants to connect to the service, it should first check for the availability of the services from the service registry. Every connection to the client will also need to have a timeout added to it, beyond which the request has to be served or it has to be retried (if configured) to another instance. This way, we can minimize the **cascading failure**.

## Security

When a client invokes an available service, we need to validate the request. In order to prevent unwanted requests from piling up, we should have an additional layer of **security**. The requests from the client should be authenticated and authorized to call the other service, to prevent unauthorized calls to the service. The service should, in turn, decrypt the request, understand whether it is valid or invalid, and do the rest.

In order to provide secure microservices, it should have the following characteristics:

- **Confidentiality:** Allow only authorized clients to access and consume the information.
- **Integrity:** Can guarantee the integrity of the information that it receives from the client and ensure that it is not modified by a third party (for example, when a gateway and a service is talking to each other, no party can tamper with, or alter, the messages that are sent between them; this a classic man-in-the-middle attack)
- **Availability:** A secure API service should be readily available.
- **Reliability:** Should handle the requests and process them reliably.



For more information on MITM, or man-in-the-middle, attacks, check out the following link: [https://www.owasp.org/index.php/Man-in-the-middle\\_attack](https://www.owasp.org/index.php/Man-in-the-middle_attack).

## Fault tolerance and failover

In a microservices architecture, there might be many reasons for a fault. It is important to handle faults or failovers gracefully, as follows:

- When the request takes a long time to complete, have a predetermined timeout instead of waiting for the service to respond.

- When the request fails, identify the server, notify the service registry, and stop connecting to the server. This way, we can prevent other requests from going to that server.
- Shut down the service when it is not responding and start a new service to make sure services are working as expected.

This can be achieved using the following:

- **Fault tolerance** libraries, which prevent cascading failures by isolating the remote instance and services that are not responding or taking a longer time than in the SLA to respond. This prevents other services from calling the failed or unhealthy instances.
- **Distributed tracing system** libraries help to trace the timing and latency of the service or system, and highlight any discrepancies with the agreed SLA. They also help you to understand where the performance bottleneck is so that you can act on this.

## Options supported by JHipster

JHipster provides options to fulfill many of the preceding concepts. The most important of them are as follows:

- JHipster Registry
- HashiCorp Consul
- JHipster gateway
- JHipster Console
- Prometheus
- JWT authentication
- JHipster UAA server

## JHipster Registry

JHipster provides **JHipster Registry** (<http://www.jhipster.tech/jhipster-registry/>) as the default **service registry**. JHipster Registry is a runtime application that all microservice applications register with and get their configuration from. It also provides additional features, such as monitoring and health check dashboards.

JHipster Registry is made up of the following:

- Netflix Eureka
- Spring Cloud Config Server

## Netflix Eureka

Eureka (<https://github.com/Netflix/eureka>) consists of the following:

- **The Eureka server:**
  - The Eureka server is a REST-based service. It is used for locating services for load balancing and failover middle tiers.
  - The Eureka server helps to load balance among the instances. They are more useful in a cloud-based environment where the availability is intermittent. On the other hand, traditional load balancers help in load balancing the traffic between known and fixed instances.
- **The Eureka client:**
  - Eureka provides a Eureka client, which makes the interaction between servers seamless. It is a Java-based client.

Eureka acts as a **middle-tier** load balancer that helps to load balance the host of middle-tier services. They provide a simple round-robin-based load balancing by default. The load balancing algorithm can be customized as needed with a wrapper.

They cannot provide sticky sessions. They also fit perfectly for client-based load balancing scenarios (as seen earlier).

Eureka has no restriction on communication technology. We can use anything, such as Thrift, HTTP, or any RPC mechanisms, for communication.

Imagine our application is in different AWS Availability Zones. We register a Eureka cluster in each of the zones that hold information about available services in that region only and start the Eureka server in each zone to handle zone failures.

All the services will register themselves to the Eureka server and send their heartbeats. When the client no longer sends a heartbeat, the service is taken out of the registry itself and the information is passed across the Eureka nodes in the cluster. Then, any client from any zone will look up the registry information to locate it and then make any remote calls. Also, we need to ensure that Eureka clusters between regions do not communicate with each other.

Eureka prefers availability over consistency. That is when the services are connected to the Eureka server and it shares the complete configuration between the services. This enables services to run even when the Eureka server goes down. In production, we have to run Eureka in a high-availability cluster for better consistency.

Eureka also has the ability to add or remove servers on the fly. This makes it the right choice for a service registry and service discovery.

## Spring Cloud Config Server

In a microservices architecture, the services are dynamic in nature. They will go down and come up based on traffic or any other configuration. Due to this dynamic nature, there should be a separate, highly available server that holds the essential configuration details that all the servers need to know.

For example, our pricing service will need to know where the registry service is and how it has to communicate with the registry service. The registry service, on the other hand, should be highly available. If, for any reason, the registry service has to go down, we will spin up a new server. The pricing service needs to communicate with the config service in order to find out about the registry service. On the other hand, when the registry service is changed, it has to communicate the changes to the config server, which will then cascade the information to all the necessary services.

Spring Cloud Config Server (<https://github.com/spring-cloud/spring-cloud-config>) provides the server- and client-side support for external configuration.

With the Spring Cloud Config Server, we have a central place to manage all our external properties across all environments. The concept is similar to Spring-based environment property source abstractions on both the client and server. They fit any application running in any language.

They are also helpful for carrying the configuration data between various (development/test/production) environments and can also help to migrate applications.

Spring Cloud Config Server has an HTTP, resource-based API for external configuration. They will encrypt and decrypt property values. They bind to the config server and initialize a Spring environment with remote property sources. The configuration can be stored in a Git repository or in a filesystem.

## HashiCorp Consul

**Consul** (<https://www.consul.io/>) is primarily a service discovery client from HashiCorp. It focuses on consistency. Consul is entirely written in Golang.

This means it will have a lower memory footprint. Added to that, we can also use Consul with services written in any programming language.

The main advantages of using Consul are as follows:

- It has a lower memory footprint.
- It can be used with services that are written in any programming language.
- It focuses on consistency rather than availability.

Consul also provides service discovery, failure detection, multi-datacenter configuration, and key-value storage.



This is an alternative option to JHipster Registry. There is an option to choose between JHipster Registry and Consul during application creation.

Eureka (JHipster Registry) requires each application to use its APIs for registering and discovering themselves. It focuses on availability over consistency. It supports only applications or services written in Spring Boot.

On the other hand, Consul runs as an agent in the services and checks the health information and a few other extra operations listed previously.

## Service discovery

Consul can attach to a service and other clients can use Consul to discover the providers of a given service. Using either DNS or HTTP, applications can easily find the services that they depend on.

## Health discovery

Consul clients can provide any number of health checks, either associated with a given service or with the local node. This information can be used by a health check service to monitor services' health, and is, in turn, used to discover the service components and route traffic away from unhealthy hosts and toward healthy hosts.

## Key/Value store

Consul has an easy-to-use HTTP API that makes it simple for applications to use Consul's Key/Value store for dynamically configuring services, electing the leader when the current leader goes down, and segregating containers based on features.

## Multiple data centers

Consul supports multiple data centers out of the box. This means you do not have to worry about building additional layers of abstraction to grow to multiple regions.

Consul should be a distributed and highly available service. Every node that provides services to Consul runs a consul agent, which is mainly responsible for health checking. These agents will then talk with one or more Consul servers, which collect and add this information. These servers will also elect a leader among themselves.

Thus, Consul serves as a service registry, service discovery, health check, and K/V store.

## JHipster gateway

In a gateway-driven microservices architecture, we need an entry point to access all the running services. So we need a service that acts as a gateway or edge service. This will proxy or route clients' requests to the respective services. In JHipster, we provide JHipster gateway for that.



JHipster gateway is a microservice application that can be generated. It integrates Netflix Zuul and Hystrix in order to provide routing, filtering, security, circuit breaking, and so on.

## Netflix Zuul

In a microservices architecture, Zuul is a front door for all the requests (gatekeeper). It acts as an edge service application. Zuul is built to enable *dynamic routing, monitoring, resiliency, and security* among the services. It also has the ability to dynamically route requests as needed.

Trivia: In *Ghostbusters*, Zuul is the gatekeeper.



Zuul works based on different types of filters that enable us to quickly and nimbly apply functionality to our edge service.

These filters help us to perform the following functions:

- **Authentication and security:** To identify each resource's authentication requirements and to reject requests that do not satisfy the requirements
- **Insights and monitoring:** To track data and statistics at the edge and to give an insight into the production application
- **Dynamic routing:** To dynamically route requests to different backend clusters as needed, based on health and other factors
- **Multi-regional resiliency (AWS):** To route requests across AWS regions in order to diversify our Elastic Load Balancer usage and move our edge closer to our members

For more information on Zuul, please check out <https://github.com/Netflix/zuul/wiki>.



## Hystrix

Hystrix (<https://github.com/Netflix/Hystrix>) is a latency and fault tolerance library designed to isolate points of access to remote systems, services, and third-party libraries. It can stop cascading failures and enable resilience in complex distributed systems where failure is inevitable.

Hystrix is designed to do the following:

- Stop cascading failures in a complex distributed system
- Protect the system from the failures of dependencies over the network
- Control the latency of the system
- Recover rapidly and fail faster to prevent cascading
- Fall back and gracefully degrade when possible
- Enable near-real-time monitoring, alerting, and operational control

When you have an application in a complex distributed architecture with a lot of dependencies, some of these dependencies will inevitably fail at some point. If your application is not isolated from these external failures, there is the risk of the application going down along with the dependency.

Hystrix is currently in maintenance mode and future versions of JHipster might migrate to a different library for fault-tolerance and circuit-breaking.

## JHipster Console

JHipster Console (<https://github.com/jhipster/jhipster-console>) is a monitoring solution for microservices built using the Elastic (ELK) Stack and Zipkin. It comes bundled with preset dashboards and configurations. It is provided as a runtime component in the form of a Docker image.

The Elastic Stack is made up of Elasticsearch, Logstash, and Kibana.



Logstash can be used to normalize the data (usually from logs), and then Elasticsearch is used to process the same data faster. Finally, Kibana is used to visualize the data.

## Elasticsearch

Elasticsearch is a widely used search engine in data analytics. It helps you to extract data really fast from data haystacks. It also helps to provide real-time analytics and data extraction. It is highly scalable, available, and multi-tenanted.

It also provides full text-based searches saved as a document. These documents, in turn, will be updated and modified based on any changes to the data. This, in turn, will provide a faster search and analysis of the data.

## Logstash

Logstash (<https://www.elastic.co/products/logstash>) will take the logs, process them, and convert them into data points. They can read any type of logs, such as system logs, error logs, and app logs. They are the **heavy working** component of this stack, which helps to store, query, and analyze the logs.

They act as a pipeline for event processing and are capable of processing huge amounts of data with the filters and, along with Elasticsearch, deliver results really fast. JHipster makes sure that the logs are in the correct format so that they can be grouped and visualized in the correct way.

## Kibana

Kibana (<https://www.elastic.co/products/kibana>) forms the frontend of the ELK Stack. It is used for data visualization. It is merely a log data dashboard. It is helpful in visualizing the trends and patterns in data that are otherwise tedious to read and interpret. It also provides an option to share/save, which makes visualization of the data more useful.

## Zipkin

Zipkin (<https://zipkin.io/>) is a distributed tracing system. Microservices architecture always has latency problems, and a system is needed to troubleshoot the latency problem. Zipkin helps to solve the problem by collecting timing data. Zipkin also helps to search the data.

All registered services will report timing data to Zipkin. Zipkin creates a dependency diagram based on the received traced requests for each of the applications or services. Then, it can be used to analyze, spot an application that takes a long time to resolve, and fix it as needed.

When a request is made, the trace instrumentation will record tags, add the trace headers to the request, and finally record the timestamp. Then, the request is sent to the original destination and the response is sent back to the trace instrumentation, which then records the duration and shares the result with the Zipkin collector, which is responsible for storing the information.

By default, JHipster will generate the application with Zipkin disabled, but this can be enabled in the `application-<env>.yml` file.

## Prometheus

In a microservices architecture, we need to monitor our services continuously and any issues should cause alerts immediately. We need a separate service that will continuously monitor and alert us whenever something weird happens. Prometheus is an alternative monitoring solution we can use.



Prometheus, along with Grafana, can be an alternative to JHipster Console. It provides monitoring and alerting support. This requires running a Prometheus server and Grafana separately. Visit <https://www.jhipster.tech/monitoring/#configuring-metrics-forwarding> to learn about using Prometheus with JHipster. To get started with Prometheus, visit <https://prometheus.io/>.

Prometheus consists of the following:

- Prometheus server, which is responsible for scraping and storing the time series data
- Libraries to instrument the application code
- A push gateway for supporting short-lived jobs
- An exporter to Grafana to visualize data
- An alert manager
- Other support tools

Let's now look at the benefits of Prometheus:

- It provides multi-dimensional data models, which are time series and are identified by metric names and key-value pairs.
- It has a flexible dynamic query language. It supports pulling time series out of the box and pushing time series via an intermediary gateway.
- It has multiple modes of graphing and dashboard support.
- It is helpful in identifying problems when there is an outage. Since it is autonomous and does not depend on any remote services, the data is sufficient for finding where the infrastructure is broken.
- It is helpful in recording the time series data and monitoring either via machine or highly dynamic service-oriented architecture.

Some things to consider when choosing Prometheus over JHipster Console are as follows:

- Prometheus is very good at exploiting the metrics of your application and will not monitor logs or traces. JHipster Console, on the other hand, uses the Elastic Stack and monitors the logs, traces, and metrics of your application.
- Prometheus can be used to query a huge amount of time series data. ELK on JHipster Console is much more versatile in terms of tracking and searching the metrics and logs.
- JHipster Console uses Kibana to visualize the data, while Prometheus uses Grafana (<https://grafana.com/>) to visualize the metrics.

## JWT authentication

We need to transfer information between microservices securely. The requests must be verified and signed digitally, where the applications verify the authenticity of the requests and respond to them.

We need to have a compact way to handle this information in the REST or HTTP world since the information is required to be sent with each request. JWT is one of the options here. JWT (JSON web tokens) is an open web standard that helps to securely transfer information between parties (applications). JWT will be signed using a secret, based on the HMAC algorithm, or with a public/private key. They are compact and self-contained:

- **Compact:** They are small and can be sent in each request.
- **Self-contained:** The payload contains all the necessary details about the user, which prevents us from querying the database for user authentication.



For advanced use cases, we can add Bouncy Castle ([librariesio](https://libraries.io/))[https://en.wikipedia.org/wiki/Bouncy\\_Castle\\_\(cryptography\)](https://en.wikipedia.org/wiki/Bouncy_Castle_(cryptography))-based encryption.

JWT consists of the header, payload, and signature. They are Base64-encoded strings, separated by a . (a period):

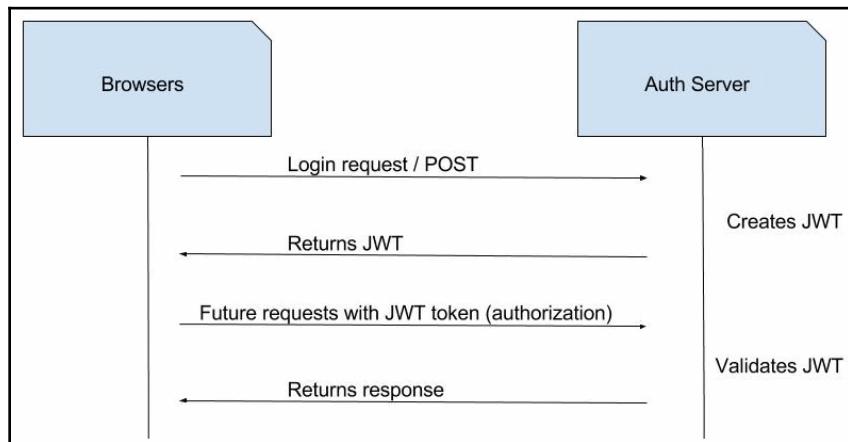
```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IlNlbmRpBCBLdW1hciBOIiwiYWRtaW4iOnRydWV9.ILwKeJ128TwDzmLGAeeY7qiROxA3kXixOG4MxTQVk_I
```

```
#Algorithm for JWT generation
HMACSHA256(
    base64UrlEncode(header) + "." +
    base64UrlEncode(payload),
)
```

Here, the HMAC SHA256 (<https://en.wikipedia.org/wiki/HMAC>) algorithm is used to encode the header and payload of the JWT.

## How JWT works

When a user logs in to the system, a token is generated based on the payload (that is, the user information and secret key). The generated token is stored locally. For all future requests, this token is added to the request and the application will validate the token before responding to the request:



The token will be in this format:

```
Authorization: Bearer <token>
```

In JHipster, we use **JJWT** (short for **Java-based JSON Web Tokens**) from Okta. This is a simplified builder pattern-based library used to generate and sign the token as a producer and parse and validate the token as a consumer.

## JHipster UAA server

JHipster **user accounting and authorizing (UAA)** server is merely an OAuth2 server that can be used for *centralized identity management*. In order to access the protected resource and also to avoid unwanted access to the APIs, there has to be an authorization server that authorizes the request and provides access to the resource.

OAuth2 is an authorization framework that provides access to the request based on tokens. Clients request access to a service; if the user is authorized, the application receives an authorization grant. After receiving the grant, the client requests a token from the authorization server. Once the token is received, the client will then request that the resource server gets the necessary information.

JHipster supports both standard LDAP protocols and is invoked via JSON APIs.



JHipster UAA is a centralized server for user accounting and authorizing service for securing JHipster microservices using the OAuth2 authorization protocol. They also have session-related information and role-based access control with the help of a user and role management that is available inside the system.

JHipster UAA is a JHipster-generated application consisting of user and role management. It also has a full-fledged OAuth2 authorization server. This is flexible and completely customizable.

Security is essential in a microservices architecture. The following are the basic requirements for securing microservices:

- They should be authenticated in one place. Users should experience the entire experience as a single unit. Once the end user logs in to the application, they should be able to access whatever they have access to. They should hold session-related information throughout the time they are logged in to the system.
- The security service should be stateless. Irrespective of the service, the security service should be capable of providing authentication for requests.
- They also need to have the ability to provide authentication to machines and users. They should be able to distinguish between them and trace them. Their function should be authorizing the incoming request rather than identifying the end user.
- Since the underlying services are scalable, security services should also have the ability to scale up and down based on requirements.
- They should, of course, be safe from attacks. Any known vulnerability should be fixed and updated as and when required.

The previous requirements can be met by using the OAuth2 protocol. The OAuth2 protocol, in general, provides the token for authenticating based on the details provided, which makes them stateless and able to authenticate a request from any source.

# Summary

So far, we have seen the benefits of a microservices architecture over monolithic applications. We also learned about the components that we need to run a microservice application such as JHipster Registry, Consul, Zuul, Zipkin, the Elastic Stack, Hystrix, Prometheus, JWT, and the JHipster UAA server.

In our next chapter, we will see how to build microservices using JHipster. We will also learn how we can choose the previous components and how easy it is to set them up with JHipster.

# 9

# Building Microservices with JHipster

So far in this book, we have generated, developed, and deployed a monolithic application using JHipster. Now it's time to build a full-fledged microservices stack. In the previous chapter, we saw the benefits that are offered by a microservice stack. In this chapter, we will look at how to build microservices with JHipster.

We will start by converting our monolithic store application into a microservice gateway application. Next, we will add new functionality to our e-commerce shop as separate microservice applications. By doing this, we can develop our microservices further so that we can include our domain model and additional business logic.

Since we are converting our online shop monolith into a microservice architecture, we will also learn how the domain model we created using **JHipster Domain Language (JDL)** can be converted into a microservice domain model.

We will then see how these applications communicate with each other and work as a single application for our end users.

In this chapter, we will cover the following topics:

- A brief introduction to application generation with JDL
- Modeling a gateway application
- Modeling the microservice applications
- Creating the domain model using JDL
- Running through the generated code

## Application architecture

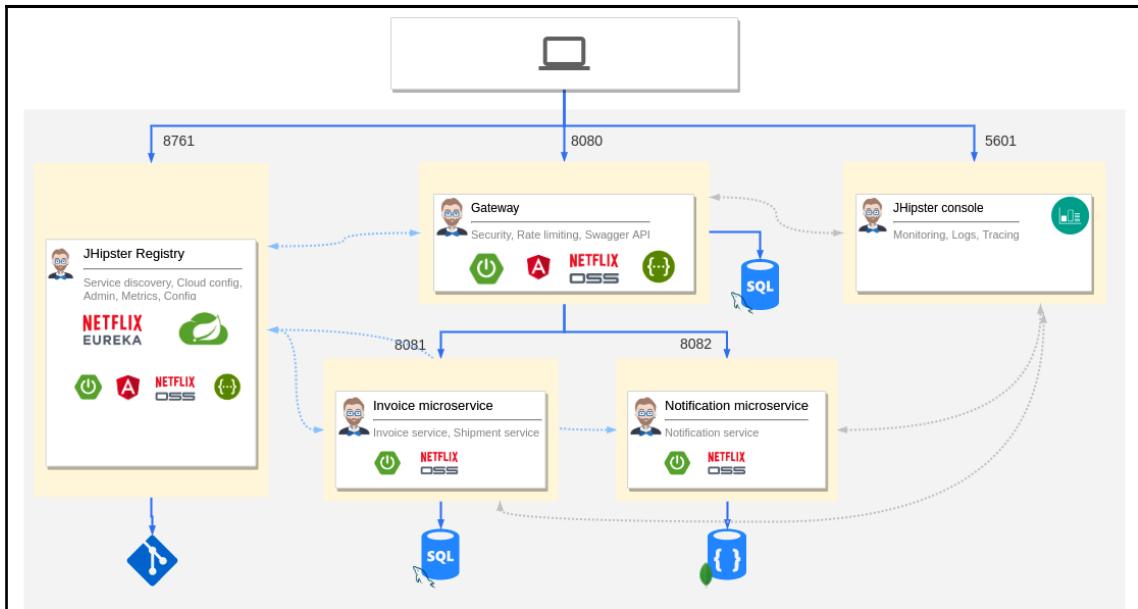
We built an online e-commerce shop using JHipster in Chapter 3, *Building Monolithic Web Applications with JHipster*. It was built as a monolith since the scope was small and it was an easier choice to start with. Let's say that our e-commerce store has grown tremendously in terms of users and scope, resulting in a more demanding situation. The team is finding it difficult to roll out features faster with the monolithic architecture and would like to have more control over individual parts of the application.

One of the solutions to this problem would be to adopt a microservice architecture. The application was created using JHipster; the option to move to microservices is much easier to accomplish. JHipster follows the **proxy microservice pattern** in which there is an aggregator/proxy in front of the services, that acts as the gateway for the end users. In much simpler terms, JHipster creates a gateway (an edge service that handles all the user requests) and the individual services that talk via the gateway to the users.

This being said, we need to have a gateway service, along with one or a few microservice applications that can run independently.

Our customers are facing some issues regarding invoicing since it is taking longer for the system to respond. Customers are also complaining that they are not receiving notifications so that they can track their orders. To solve this, we will remove the invoice service from our monolithic application and make it a separate service, and then create a separate **notification service** that will take care of the notifications. For the former, we will stick with the same SQL database. For the latter, we will use the NoSQL database.

Let's have a look at the application architecture that we are going to generate:



As you can see, we have a gateway application running on port 8080, which will be the user-facing application. Then, we have two microservice applications with their own databases. The gateway will route requests to these services via ports 8081 and 8082. There is also JHipster Registry running on port 8761 providing service discovery and config management.

In the next section, we'll learn how to create this architecture using JHipster.

## Generating a microservice stack using JDL

We will be using the JDL to generate our microservice applications. Before we prepare the JDL, let's take a quick look at the language definition for creating applications. The official documentation for JDL can be found at <https://www.jhipster.tech/jdl/>.

## Application modeling using JDL

In Chapter 4, *Entity Modeling with the JHipster Domain Language*, we learned how to define entities and relationships using JDL. Now, let's learn how to define applications.

In the following syntax, take note of the following:

- [] denotes **optional**.
- \* denotes **more than one can be specified**.

JavaDocs can be added to entity declarations, while `/** */` Java comments can be added to fields and relationship declarations. JDL-specific comments can only be added using the `//` syntax.

It is also possible to define numerical constants in JDL, for example, `DEFAULT_MIN_LENGTH = 1.`

The application declaration is created using the following syntax:

```
application {
    config {
        <option> <value>
    }
    [entities <entity names(comma separated)>* [except] [<entity
        names(comma separated)>*] ]
}
```

`<option>` is one of the supported application options, followed by `<value>`, which is supported by the option. If an option is not declared, a default value will be used instead. Refer to the table at <https://www.jhipster.tech/jdl/applications#available-application-options> for all the supported options and defaults.

The `entities` keyword is used to associate the entities applicable to a particular application. `entities *` will denote that all the entities declared in the file are associated with an application. Fine-grained declarations are possible using the given syntax; for example, `entities * except C, D` or `entities A, B`.

An example of an application declaration is as follows:

```
application {
    config {
        baseName myapp
        applicationType microservice
        prodDatabaseType postgresql
        buildTool gradle
    }
}
```

With entities, it would look as follows:

```
application {
    config {
        baseName myMonolith
        applicationType monolith
    }
    entities * except C, D
}

entity A
entity B
entity C
entity D
```

Let's proceed and design our JDL.

## Gateway application

Even though microservices are made up of different services, for end users, they should be a single, unified product. There are a lot of services that are designed to work in a lot of different ways, but there should be a single entry point for users. Thus, we need a gateway application since they form the frontend of your application. This is called the proxy pattern or gateway pattern.

They segregate internal contracts and services from external users. We may have application-level internal services that we shouldn't expose to external users, so these can be masked away. This also adds another level of security to the application.

We can use JDL to convert an existing monolithic application into a microservice gateway. If you want to create everything from scratch, skip the following note and just follow along.



If you want to use the existing monolith application we created in the previous chapters and convert that, then make sure the application was created inside a folder such as `e-commerce-app`. Rename the `online-store` application folder to `store`. Then, navigate to the `store` folder and commit the changes. Now, create a new Git branch with `git checkout -b microservice-conversion` so that we can do a clean merge back to the master once we are done.

We will start by converting the monolithic application configuration that we have generated into a microservice gateway application with JDL.

## JDL specification for the gateway application

We have already generated our monolithic application, as well as our entities. As a part of the monolithic application's generation, we have selected some options via JHipster CLI. We will stick to the same options (the database, authentication type, package name, i18n, and so on) when we design the microservice gateway application using JDL.



We will learn how the customizations that we applied in the monolithic application can be applied to the gateway later.

It's coding time now, so let's start building a gateway application using JDL.

The first step is to convert the monolithic application into a microservice gateway application with almost the same configuration that we used when we created a monolithic application.



You can also get the JDL of the existing application to start with by running `jhipster export-jdl app.jdl`, which will include the application JDL.

Open your favorite IDE, text editor, or JDL-Studio (<https://start.jhipster.tech/jdl-studio/>) and start with a file – let's call it `app.jdl`. We will save this file in the `e-commerce-app` folder, alongside our store app.

The JDL will look as follows:

```
application {
    config {
        baseName store
        applicationType gateway
        packageName com.mycompany.store
        serviceDiscoveryType eureka
        authenticationType jwt
        prodDatabaseType mysql
        cacheProvider hazelcast
        buildTool gradle
        clientFramework angularX
        useSass true
        languages [en, zh-cn, ta]
        websocket spring-websocket
        testFrameworks [protractor]
    }
}
```

Since we are working with microservices, there is a high risk of having port conflicts. In order to avoid them, we need to select a port for each microservice application (the gateway and the application). By default, we will have 8080 as the port, but we can change the port as necessary. For now, we will use the default port since the gateway will run on 8080, similar to what our monolithic application had.

We will try not to declare the default options in the JDL to keep it simple.

In order to make the monolith into a microservice gateway application, the only things we need to change are `serviceDiscoveryType` to `eureka` since, for monolithic applications, it is not mandatory to have service discovery, and `applicationType` to `gateway`.



When you select **No service discovery**, the microservice URLs are hardcoded in the property files.

For the authentication type, JHipster provides three options for the authentication type: JWT, OAuth2, and UAA server-based types. JWT is stateless, while the UAA runs on a different server (and application altogether). OAuth2, on the other hand, will provide authorization tokens, while the authorization is done on the third-party system. We will stick with JWT here.



JHipster allows you to create a UAA server application.

We will stick to the **MySQL** database for production, **Hazelcast** for `cacheProvider`, **Gradle** for `buildTool`, and **Angular X** for `clientFramework`, the same as in the monolithic store.

Then, we can set any other additional technologies that we need to use. JHipster provides us with the option to select Elasticsearch, using Hazelcast for clustered applications, WebSockets, and OpenAPI Generator for API-based development and Kafka-based asynchronous messaging. We will set WebSockets here, similar to what we used in our monolithic store.

Everything else remains in their default states since this is what we used in our monolith as well.

## Microservice invoice application

If we look at the invoice domain from our monolithic application, it can be easily separated into a separate microservice application. Let's name it **Invoice Service**. This service is responsible for creating and tracking invoices.

## JDL specification for the invoice application

For the invoice application, our JDL will look as follows. Add it after the gateway application declaration:

```
application {
    config {
        baseName invoice,
        applicationType microservice,
        packageName com.mycompany.store,
        serviceDiscoveryType eureka,
        authenticationType jwt,
        prodDatabaseType mysql,
        buildTool gradle,
        serverPort 8081,
        languages [en, zh-cn, ta]
    }
}
```

We will set `applicationType` to `microservice` and, similar to our gateway, select `jwt` for `authenticationType`, `eureka` for `serviceDiscoveryType`, and so on. Since we don't want any port conflicts, we will also set `serverPort` to 8081.

## Microservice notification application

For an e-commerce website, it is essential that orders are tracked and users are notified at the right moment. In this section, we will create a notification service that will notify users whenever their order status changes.

## JDL specification for the notification application

For the notification application, our JDL will look as follows. Add it after the invoice application declaration:

```
application {
    config {
        baseName notification,
        applicationType microservice,
        packageName com.mycompany.store,
        serviceDiscoveryType eureka,
        authenticationType jwt,
        databaseType mongodb,
        cacheProvider no,
        enableHibernateCache false,
        buildTool gradle,
        serverPort 8082,
        languages [en, zh-cn, ta]
    }
}
```

We will set `applicationType` to `microservice` and, similar to our gateway, select `jwt` for `authenticationType` and `eureka` for `serviceDiscoveryType`.

Since we have selected 8080 for the monolithic application and 8081 for the invoice service, we will use port 8082 for the notification service.

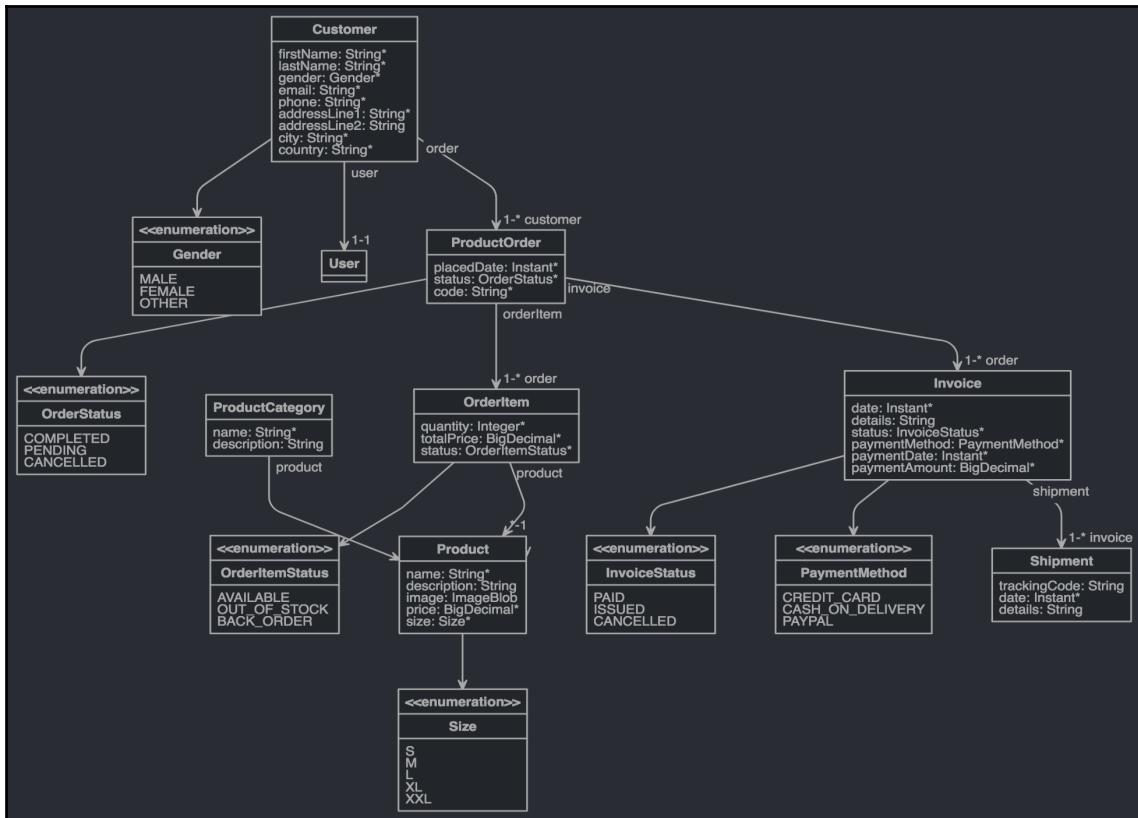
Here, we set **MongoDB** as the database, unlike our other applications. Since this option doesn't support using a level 2 cache, we will disable it by setting `cacheProvider` and `enableHibernateCache` to negative values.

## Modeling microservice entities in JDL

Since we already created a JDL for our entities when we set up our monolithic application, it's time to update and use it here.

As we discussed in the previous chapter, we will move the entities from a monolithic application to a gateway application. Then, we'll remove the invoice-related entities from the monolithic application, use them in our invoice microservice, and update the related invoice references in that. Finally, we'll create entities for the notification microservice.

The following diagram shows our JDL entity model:



As you can see, the invoice is a perfect candidate to move out into a separate service. We can completely decouple the invoice and its dependencies, but this will cause one problem in our current application – the `ProductOrder` entity is related to the `Invoice` table and we have to remove this dependency while keeping the relationship (but not as a foreign key) as an indirect key in `Invoice` that connects to the `ProductOrder` entity.

This can be achieved in two ways. We can change the foreign key into just another column in the `Invoice` entity, or we can create another entity called `InvoiceOrder` that just holds `Invoice` IDs and `ProductOrder` IDs and map it to the `Invoice` entity.

The former keeps the table structure more or less the same and allows for easier migration. The latter will increase isolation at the cost of normalization, and they are heavily used in high-performance applications. As you can see, both have their own merits and demerits. The approach you should take depends purely on your requirements.

We will consider the first approach:

1. As a first step, copy the JDL defined in `online-store.jdl` in the store application to the new `app.jdl` file that we are working with.
2. Now, we will remove the relationship from the `ProductOrder` entity, like so:

```
relationship OneToMany {  
    ...  
    ProductOrder{invoice} to Invoice{order(code) required}  
    ...  
}
```

3. Remove the highlighted line.
4. Then, go to the `Invoice` entity, add a `productOrderId` field, and mark it as the `Long` type. It is a `required` field, so it needs the `required` keyword:

```
entity Invoice {  
    code String required  
    date Instant required  
    details String  
    status InvoiceStatus required  
    paymentMethod PaymentMethod required  
    paymentDate Instant required  
    paymentAmount BigDecimal required  
    productOrderId Long required  
}
```

Entities for microservices can be tagged using the `microservice` keyword supported by JDL. This helps JHipster to identify entities that belong to a specific microservice. It follows the same JDL options syntax that we saw earlier:

```
<OPTION> <ENTITIES | * | all> [with <VALUE>] [except  
<ENTITIES>]
```

Using the preceding syntax, we need to write the following:

- The `microservice` keyword.
- Followed by the names of the entity; these have to be comma-separated if multiple entities are used.
- Followed by the `with` keyword.
- Followed by the name of the microservice; for example, `Invoice`.

An example of this is `microservice Invoice, Shipment with invoice`.



You can use different files for the microservice entities if you like so that we create two files, `invoice-jdl.jh` and `notification-jdl.jh`, that contain the entities related to invoice and notification, respectively, along with the original. When importing, we can import multiple files.

5. Then, we map the existing `Invoice` entity to the microservice in our JDL:

```
/* Entities for Invoice microservice */
entity Invoice {
    code String required
    date Instant required
    details String
    status InvoiceStatus required
    paymentMethod PaymentMethod required
    paymentDate Instant required
    paymentAmount BigDecimal required
}

enum InvoiceStatus {
    PAID, ISSUED, CANCELLED
}

entity Shipment {
    trackingCode String
    date Instant required
    details String
}

enum PaymentMethod {
    CREDIT_CARD, CASH_ON_DELIVERY, PAYPAL
}

relationship OneToMany {
    Invoice{shipment} to Shipment{invoice(code) required}
}

service Invoice, Shipment with serviceClass
paginate Invoice, Shipment with pagination
microservice Invoice, Shipment with invoice
```

6. Now, it is time to create notification service entities. Add the entities for the notifications, as follows:

```
/* Entities for notification microservice */
entity Notification {
    date Instant required
    details String
```

```
        sentDate Instant required
        format NotificationType required
        userId Long required
        productId Long required
    }

    enum NotificationType {
        EMAIL, SMS, PARCEL
    }

microservice Notification with notification
```

We have bound these entities to the `Notification` microservice.

Now, we need to let the applications know which entities to include, as follows:

```
application {
    config {
        baseUrl store
        ...
    }
    entities *
}

application {
    config {
        baseUrl invoice,
        ...
    }
    entities Invoice, Shipment
}

application {
    config {
        baseUrl notification,
        ...
    }
    entities Notification
}
```

We have declared `entities *` for the gateway as it will need to generate the frontends for all our entities.

- Finally, retain the general options as well:

```
service * with serviceClass
paginate Product, Customer, ProductOrder, Invoice, Shipment,
OrderItem with pagination
```

Save this file as `app.jdl` in the `e-commerce-app` folder we created and commit it to Git.

That's it. We have defined the domain model for our microservices. The final JDL can be found at <http://bit.ly/jh-book-jdl>.

## Application generation with import-jdl

In this section, we'll generate our applications using the JDL we just created.

Run the following command in the `e-commerce-app` folder:

```
> cd e-commerce-app
// delete the store folder so that we have better diff when
// new app is generated
> rm -rf store
> jhipster import-jdl app.jdl --skip-git
```



The `--skip-ui-grouping` flag can be used to disable the client-side entity-component grouping behavior for microservices that were introduced in JHipster 5.x. This grouping behavior is useful when you have entities with the same name in different services.

JHipster will start creating applications and entities in a parallel process to speed things up. If you would like the process to be more interactive so that JHipster will ask about overwriting the modified files, pass the `--interactive` flag to the preceding command. JHipster then asks whether you want to overwrite the conflicting files or use your existing ones, as well as a few other options. Users can use any one of the desired options. We will use "a", which means that it will overwrite everything.



This prompt is extremely useful if you have a lot of custom code written on your application. You can choose the appropriate option to get the desired result. You can also use Git for the same purpose.

If you had the monolithic application in the `e-commerce-app` folder, this will overwrite all the customizations we made in our monolithic application. We can easily bring them back into this branch by cherry-picking the required changes from our master branch using Git. You can follow a similar approach to the one we looked at in [Chapter 5, Customization and Further Development](#), for that. Once all the changes have been applied, we can merge this branch back into the master. You will have to do the same for any entity files that have been modified as well:

```
Entity Product generated successfully.  
Entity ProductCategory generated successfully.  
Entity Customer generated successfully.  
Entity ProductOrder generated successfully.  
Entity OrderItem generated successfully.  
Entity Invoice generated successfully.  
Entity Shipment generated successfully.  
Entity Notification generated successfully.  
INFO! Congratulations, JHipster execution is complete!  
INFO! App: child process exited with code 0
```

You will see the preceding output when our microservice application is generated. JHipster will automatically commit the generated files to Git. If you wish to do this step yourself, you can do so by providing the `skip-git` flag during execution, for example, with the help of the `jhipster import-jdl app.jdl --skip-git` command, like we've done here, and executing the steps manually.

Don't forget to commit the changes in each of the services and gateways. You could also init the entire `e-commerce-app` folder as a git source if you like by running `git init`:

```
> cd e-commerce-app  
> git init  
> git add invoice/  
> git commit -m "invoice application generated using JDL"  
> git add notification/  
> git commit -m "notification application generated using JDL"  
// Manually compare and unstage files from store that you do not want  
// overridden  
// Cherry-pick required changes done earlier  
> git add --all  
> git commit -m "store application converted to gateway using JDL"
```

If something goes wrong during cherry-picking, don't worry – you can always copy the `app.jdl` file to a new folder and run the `jhipster import-jdl` command to generate fresh applications and compare them to see what went wrong.



We won't need the code changes we made for Heroku deployment anymore so we can safely override them. The security changes we made for the `Invoice` entities will also need to be refactored a bit since the relationships have changed now. This will be one of your tasks for the next steps to pursue after finishing the book.

Next, we will take a look at the generated code.

## Gateway application

The gateway application is generated in a similar fashion to the monolithic application, except for configurations related to Zuul proxy, Eureka Client, and Hystrix:

```
@SpringBootApplication
@EnableConfigurationProperties({LiquibaseProperties.class,
ApplicationProperties.class})
@EnableDiscoveryClient
@EnableZuulProxy
public class StoreApp implements InitializingBean {
    ...
}
```

We have selected the JHipster registry for our registry service. This will be a standalone registry server that other microservice applications and gateways will register:

- `@EnableDiscoveryClient` is added to Spring Boot's main class, which will enable Netflix Discovery Client. The microservice applications and gateways need to register themselves to the registry service. It uses Spring Cloud's discovery client abstraction to interrogate its own host and port and then adds them to the registry server.
- Zuul, on the other hand, is the gatekeeper. This helps route the authorized requests to the respective endpoints, limits the requests per route, and relays the necessary tokens to the microservice application.
- `@EnableZuulProxy` helps the microservice gateway application route the requests to the applicable microservice application based on the configurations provided in the `application.yml` file:

```
zuul: # those values must be configured depending on the
      application specific needs
      sensitive-headers: Cookie, Set-Cookie
      host:
        max-total-connections: 1000
        max-per-route-connections: 100
```

```
prefix: /services
semaphore:
    max-semaphores: 500
```

In the gateway app, we have specified the aforementioned settings for the Zuul configuration. The maximum number of total connections that a proxy can hold open is kept at 1000. The maximum number of route connections that a proxy can hold open is kept at 100. The semaphore is kept to a maximum of 500. (Semaphore is like a counter that is used for synchronization between threads and processes.)

Zuul filters are defined under

`store/src/main/java/com/mycompany/store/gateway/` and include access control, rate limit, token relay, and a response rewriting filter.

Access to the backend microservice endpoint is controlled by `AccessControlFilter.java`, which will check whether the request is authorized and is allowed to request the endpoint:

```
public class AccessControlFilter extends ZuulFilter {
    ...
    public boolean shouldFilter() {
        ...
        for (Route route : routeLocator.getRoutes()) {
            ...
            if (requestUri.startsWith(serviceUrl.substring(0,
                serviceUrl.length() - 2))) {
                return !isAuthorizedRequest(serviceUrl, serviceName,
                    requestUri);
            }
        }
        return true;
    }
    ...
}
```

A rate-limiting filter is added to the generated application, which is defined in `RateLimitingFilter.java`, and limits the number of HTTP calls that are made per client. This is enabled conditionally with the following:

```
public class RateLimitingFilter extends ZuulFilter {
    ...
}
```

SwaggerBasePathRewritingFilter.java is also used, which will help us rewrite the microservice Swagger URL base path:

```
public class SwaggerBasePathRewritingFilter extends SendResponseFilter {  
    ...  
    public Object run() {  
        RequestContext ctx = RequestContext.getCurrentContext();  
        ...  
        if (context.getResponseGZipped()) {  
            try {  
                context.setResponseDataStream(new  
                    ByteArrayInputStream(...));  
            } catch (IOException e) {  
                log.error("Swagger-docs filter error", e);  
            }  
        } else {  
            context.setResponseBody(rewrittenResponse);  
        }  
    }  
    ...  
}
```

TokenRelayFilter.java is added to remove the authorization from Zuul's ignored headers list. This will help us propagate the generated authorization token:

```
public class TokenRelayFilter extends ZuulFilter {  
    ...  
    public Object run() {  
        RequestContext ctx = RequestContext.getCurrentContext();  
        Set<String> headers = (Set<String>) ctx.get("ignoredHeaders");  
        // JWT tokens should be relayed to the resource servers  
        headers.remove("authorization");  
        return null;  
    }  
    ...  
}
```

Each application should have a Eureka client that helps load balance the requests among the services, as well as send health information to the Eureka Server or registries. The Eureka client is configured in application.yml as follows:

```
eureka:  
  client:  
    enabled: true  
    healthcheck:  
      enabled: true  
    fetch-registry: true  
    register-with-eureka: true
```

```
instance-info-replication-interval-seconds: 10
registry-fetch-interval-seconds: 10
instance:
  appname: store
  instanceId: store:${spring.application.instance-id:${random.value}}
  lease-renewal-interval-in-seconds: 5
  lease-expiration-duration-in-seconds: 10
  status-page-url-path: ${management.endpoints.web.base-path}/info
  health-check-url-path: ${management.endpoints.web.base-path}/health
  metadata-map:
    zone: primary # This is needed for the load balancer
    profile: ${spring.profiles.active}
    version: #project.version#
    git-version: ${git.commit.id.describe:}
    git-commit: ${git.commit.id.abbrev:}
    git-branch: ${git.branch:}
```

We have chosen to enable health checks and have set the interval within which to register and replicate services to 10 seconds, as well as instances where we define the lease renewal interval and expiration duration.

We will configure a timeout in Hystrix, beyond which the server is considered to be closed:

```
hystrix:
  command:
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 10000
```

If the server does not respond within 10 seconds, then the server is considered dead and is unregistered from the registry service.

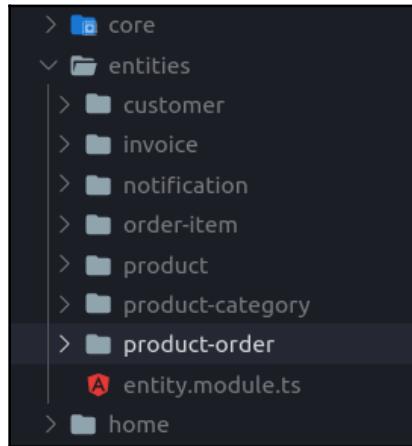


With JHipster, you can also build microservice applications that serve REST endpoints without any GUI.

This ensures no subsequent requests are sent to that server until the server is made active.

## Gateway application entities

In the gateway application, the entire frontend (including the entities in the microservices) will be generated. Since JHipster produces proxy-based microservices, all the frontend code will live in the gateway application:



`ProductOrder.java` will remove `Invoice` as a foreign key as well in comparison to our monolithic application.

Let's run all the tests to ensure we haven't broken anything from the cherry-picking we have done.

Navigate to the `store` application folder and run `./gradlew test integrationTest npm_test`; it should pass.

Next, we will look at the invoice microservice configurations.

## Invoice microservice configuration

The application that is generated will not contain any frontend code. Again, the invoice service is a Spring Boot-based application. The security features are configured in `SecurityConfiguration.java`.

It ignores all the H2 Database-related requests:

```
public void configure(WebSecurity web) {  
    web.ignoring()  
        .antMatchers("/h2-console/**");  
}
```

Since the services are independent, they can be deployed and run on another server with a different IP address. This requires us to disable **Cross-Site Request Forgery (CSRF)** by default.

We will also use the **STATELESS** session policy in session management. This is the strictest session policy available. This will not allow our application to generate a session, so our requests have to have the (time-bound) tokens attached to each and every request. This enhances the security of our services. Their stateless constraint is another advantage of using REST APIs.



For more options and information on session policies, please look at the following documentation: <https://docs.spring.io/autorepo/docs/spring-security/4.2.3.RELEASE/apidocs/org/springframework/security/config/http/SessionCreationPolicy.html>.

All the API-related requests and Swagger resources should be allowed once the request has been authorized (based on the JWT token):

```
public void configure(HttpSecurity http) throws Exception {  
    // @formatter:off  
    http  
        .csrf()  
        .disable()  
        .exceptionHandling()  
        .authenticationEntryPoint(problemSupport)  
        .accessDeniedHandler(problemSupport)  
    .and()  
        .headers()  
        .contentSecurityPolicy(...)  
    .and()  
        .referrerPolicy(...)  
    .and()  
        .featurePolicy(...)  
    .and()  
        .frameOptions()  
        .deny()  
    .and()  
        .sessionManagement()  
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
```

```
.and()
    .authorizeRequests()
        .antMatchers("/api/authenticate").permitAll()
        .antMatchers("/api/**").authenticated()
        .antMatchers("/management/health").permitAll()
        .antMatchers("/management/info").permitAll()
        .antMatchers("/management/prometheus").permitAll()
        .antMatchers("/management/**").hasAuthority
            (AuthoritiesConstants.ADMIN)
    .and()
        .apply(securityConfigurerAdapter());
// @formatter:on
}
```

On the resource side, we have a new `bootstrap.yml` file that overrides the `application.yml` files. We have also defined the registry-related information and some Spring properties here.

Our current microservice application uses the JHipster registry as the registry service in order to register and deregister their existence with a heartbeat signal. We need to provide the password of our registry service so that the application can connect to the registry service:

```
jhipster:
  registry:
    password: admin
```

Also, the name of the Spring Boot service and the default Spring Cloud Config parameters are specified in the `bootstrap.yml` file. We have also added the URI that we have to connect in order to fetch the configuration of the registry service:

```
spring:
  application:
    name: invoice
  ...
cloud:
  config:
    # if not in "prod" profile, do not force to use Spring
    # Cloud Config
    fail-fast: false
    uri: http://admin:${jhipster.registry.password}
        @localhost:8761/config
    # name of the config server's property
    # source (file.yml) that we want to use
    name: invoice
  ...
```

The `bootstrap.yml` file is used in the development process, while, for production, there is a `bootstrap-prod.yml` file. Similar to the gateway, the rest of the service-related configurations are done in the `application-* .yml` files.

The Eureka configuration is exactly the same as it was in the gateway application. All the generated applications will have a similar Eureka configuration:

```
eureka:
  client:
    enabled: true
    healthcheck:
      enabled: true
    fetch-registry: true
    register-with-eureka: true
    instance-info-replication-interval-seconds: 10
    registry-fetch-interval-seconds: 10
  instance:
    appname: invoice
    instanceId: invoice:${spring.application.instance-
      id:${random.value}}
    lease-renewal-interval-in-seconds: 5
    lease-expiration-duration-in-seconds: 10
    status-page-url-path: ${management.endpoints.web.base-path}/info
    health-check-url-path: ${management.endpoints.web.base-path}/health
    metadata-map:
      zone: primary # This is needed for the load balancer
      profile: ${spring.profiles.active}
      version: #project.version#
      git-version: ${git.commit.id.describe:}
      git-commit: ${git.commit.id.abbrev:}
      git-branch: ${git.branch:}
```

The database and JPA configurations are present in the `application-[dev|prod].yml` files:

```
spring:
  profiles:
    active: dev
  ...
  datasource:
    type: com.zaxxer.hikari.HikariDataSource
    url: jdbc:h2:file:./build/h2db/db/invoice;DB_CLOSE_DELAY=-1
    username: invoice
    password:
    hikari:
      poolName: Hikari
      auto-commit: false
```

```
...
jpa:
  database-platform: io.github.jhipster.domain.util.FixedH2Dialect
  database: H2
  show-sql: true
  properties:
    ...
liquibase:
  # Remove 'faker' if you do not want the sample data to be loaded
  # automatically
  contexts: dev, faker
```

The rest of the configurations remain similar to what was generated in the gateway application, and they can be tweaked or customized based on your requirements.

Let's run all the tests to ensure we haven't broken anything from the cherry-picking we have done.

Navigate to the `invoice` application folder and run `./gradlew test integrationTest`; it should pass.

Now, we can boot up the application alongside the gateway application and registry service if required. Since the application tries to connect to the registry service first, if there is no registry service available at the specified location, then the application will not know where to connect and whom to respond to, and hence will fail.

Before we spin up the applications, let's look at the notification service with NoSQL as the backend database.

## Notification microservice configuration

We have selected similar options for both microservices. The code that will be generated will be very similar, except for the database configuration and absence of any cache configuration:

```
spring:
  profiles:
    active: dev
  ...
data:
  mongodb:
    uri: mongodb://localhost:27017
    database: notification
  ...
```

As you can see, this service only contains the backend files and not the frontend files for the microservices since they are already generated in the gateway service.

Let's run all the tests to ensure everything works.

Navigate to the notification application folder and run `./gradlew test integrationTest`; it should pass.

## Summary

In this chapter, we generated a gateway application and two microservice applications using JDL. We have shown you how easy it is to generate a microservice architecture with JHipster. We also learned how to design JHipster applications using JDL.

Before we run our application, we need to kick-start our registry server.

In the next chapter, we will learn how to run the registry server, and we will also take a look at the newly added screens.

# 10

# Working with Microservices

In the previous chapter, we created a gateway and two microservices using JHipster. Now, we'll learn how to run the stack locally for testing and further development. But before we can start, we need to set up some tools in order to work with microservices. These tools provide some of the basic necessities, such as service discovery and configuration management, that we learned about in the previous chapters. Once we set up the required tools, we can see the applications in action.

In this chapter, we will cover the following topics:

- Setting up JHipster Registry locally
- Running a generated application locally

Let's get started!

## Setting up JHipster Registry locally

We have created our gateway and two microservice applications. These microservices have two different databases. So far, it has been easy and simple to create these with JHipster. We also enabled service discovery with Eureka for the applications. This means we would have to run a service registry in order to deploy the applications.

JHipster provides two different options we have previously seen, Consul and JHipster Registry. For our use case, since we have chosen Eureka, we need to go with JHipster Registry. We learned about JHipster Registry in [Chapter 8, \*Microservice Server-Side Technologies\*](#). Now, we will learn how to set up and start it in our local development environment.

These three services basically act as Eureka clients. We need a service registry that registers and deregisters the application as and when the application is started and stopped, respectively; this is JHipster Registry. The Eureka server (JHipster Registry server) acts as a master to all the Eureka clients.

All the microservice applications and the gateway will register/deregister themselves with/from JHipster Registry when the applications start and stop.

Let's recap a little bit of what we've learned already. The JHipster Registry is made up of a Eureka server and Spring Cloud Config Server and they help in doing the following:

- **The Eureka server** helps with service discovery and load balancing the requests.
- **The Spring Cloud Config server** acts as a single place where we will manage the external properties of applications across environments.
- It also provides a dashboard for users. With this, users can manage and monitor the application.

This makes JHipster Registry an ideal choice for both monolithic and microservice architectures.



If you are developing microservice applications where different services are written in different languages, and if you prefer consistency over availability of services, then you can choose Consul as a service discovery engine.

There are two ways in which we can set up JHipster Registry to run locally: we can either download the JAR file (pre-packaged) and run it directly, or we can use a Docker container to run it. We will learn how to do each of these now.



You can choose to use JHipster Registry while generating monolithic applications as well. Just select yes for the question **Do you want to use the JHipster Registry to configure, monitor, and scale your application?** during generation.

## Using a pre-packaged JAR file

Let's learn how to use a pre-packaged binary:

1. Download the latest version of the pre-packaged executable JAR file from the registry releases page (<https://github.com/jhipster/jhipster-registry/releases>). At the time of writing, the released version is 5.0.2.

- Once downloaded, we can run JHipster Registry using the following command:

```
> java -jar jhipster-registry-<version>.jar \
--spring.security.user.password=admin \
--jhipster.security.authentication.jwt.secret=<your-base64-
encoded-secret-key> \
--spring.cloud.config.server.composite.0.type=native \
--spring.cloud.config.server.composite.0.search-
locations=file:/central-config
```

Note that we pass a few values to our registry server; these are as follows:

```
--security.user.password=admin
```

Since JHipster Registry is built on top of the JHipster application, it will have the default admin user. For that admin user, we provide the password with the Spring security.user.password property:

```
--jhipster.security.authentication.jwt.secret=<your-base64-encoded-
secret-key>
```

Then, we define the JWT secret for the application in one of two ways: we can either set the information in the environment variable and use that, or we can add this key value when we define the secret. Make sure your secret is Base64-encoded and is at least 256 bits in length. For development, you can use the key we generated in the .yo-rc.json files:

```
--spring.cloud.config.server.composite.0.type=native
--spring.cloud.config.server.native.search.locations=file:/central-
config
```

Finally, we tell the JHipster Registry where to find the central configurations that are available for the Spring Cloud Config Server by using the preceding spring.cloud.config properties.

Before we look at what value to pass in here, we need to know about the Spring profiles in the context of spring-cloud-config. Spring Cloud Config supports native and git profiles by default.

In a native profile, the Cloud Config server expects its properties to be defined in a file, and we have to pass in the file location to the JHipster Registry. On the other hand, the git profile will expect --spring.cloud.config.server.git.uri to be set.

For example, the sample JHipster config file for the registry is as follows:

```
configserver:
  name: JHipster Registry config server
  status: Connected to the JHipster Registry config server using ...
jhipster:
  security:
    authentication:
      jwt:
        secret: my-secret-key-which-should-be-changed
          -in-production-and-be-base64-encoded
```

This can also be seen in the Spring Cloud Configuration page of the registry if the file is loaded:

The screenshot shows the JHipster Registry interface with the Spring Cloud Configuration page loaded. At the top, there's a navigation bar with links for Home, Eureka, Configuration, Administration, and a back arrow. Below the navigation, there are buttons for Refresh now and disabled. The main area has sections for Application, Profile, and Git Label, each with input fields. Under Configuration, there are tabs for YAML, Properties, JSON, and Table, with the YAML tab selected. The configuration content is displayed in a dark box:

```
configserver:
  name: Docker JHipster Registry
  status: Connected to the JHipster Registry running in Docker
eureka:
  client:
    service-url:
      defaultZone: http://admin:${jhipster.registry.password}@localhost:8761/eureka/
jhipster:
  security:
    authentication:
      jwt:
        secret: my-secret-key-which-should-be-changed-in-production-and-be-base64-encoded
```

Below the configuration, there's a section for Configuration Sources with an Index table:

Index	Configuration
0	<ul style="list-style-type: none"><li>type: "native"</li><li>search: {"locations": "file:/central-config/localhost-config/"}</li></ul>

Just like the JHipster app provides the `dev` and `prod` profiles, JHipster Registry also supports the `dev` and `prod` profiles. By default, it will run in the `dev` profile when started, but we can make it run in a `prod` profile by providing the `--spring.profiles.active=prod`, `git` flag to the `java -jar` command. We also need to pass the `git` URL using the `--spring.cloud.config.server.git.uri` flag. For production mode, `git` is the preferred profile to use on a Spring Cloud Config Server.

## Docker mode

You can also start JHipster Registry from the Docker image provided by JHipster. This is the easiest way to run the application. The application that we generated already has the `docker-compose` file that's required.

For example, in the gateway application we created, look for the `docker-compose` file under `src/main/docker/jhipster-registry.yml`.

We can start the JHipster Registry by typing the following command in the Terminal:

```
> docker-compose -f store/src/main/docker/jhipster-registry.yml up
```

The `docker-compose` file (`src/main/docker/jhipster-registry.yml`) contains the following code:

```
version: '2'
services:
  jhipster-registry:
    image: jhipster/jhipster-registry:v5.0.2
    volumes:
      - ./central-server-config:/central-config
    environment:
      - _JAVA_OPTIONS=-Xmx512m -Xms256m
      - SPRING_PROFILES_ACTIVE=dev,swagger
      - SPRING_SECURITY_USER_PASSWORD=admin
      - JHIPSTER_REGISTRY_PASSWORD=admin
      - SPRING_CLOUD_CONFIG_SERVER_COMPOSITE_0_TYPE=native
      -
    SPRING_CLOUD_CONFIG_SERVER_COMPOSITE_0_SEARCH_LOCATIONS=file:./central-
config/localhost-config/
    ports:
      - 8761:8761
```

This defines the image as `jhipster-registry` with a version (the latest). It also defines a volume that's used to mount `central-config`, which is required by the Spring Cloud Config Server to define the application properties for the microservice application and gateway. The environment variables, such as the Spring profile, password for the admin, and cloud-config search location, are also defined here. The port in which it is exposed (8761) is also specified. The config for production is also present in the same file but commented out.

To do this, you need Docker installed and running on your machine.



In all the preceding cases (when they are successful), it boots up JHipster Registry on port 8761 and uses native mode by default (unless otherwise changed explicitly). You can navigate to `http://localhost:8761` to access JHipster Registry and then log in to the application with the password that we used when we started the application.

Now, let's learn how we can run our microservice stack locally.

## Running a generated application locally

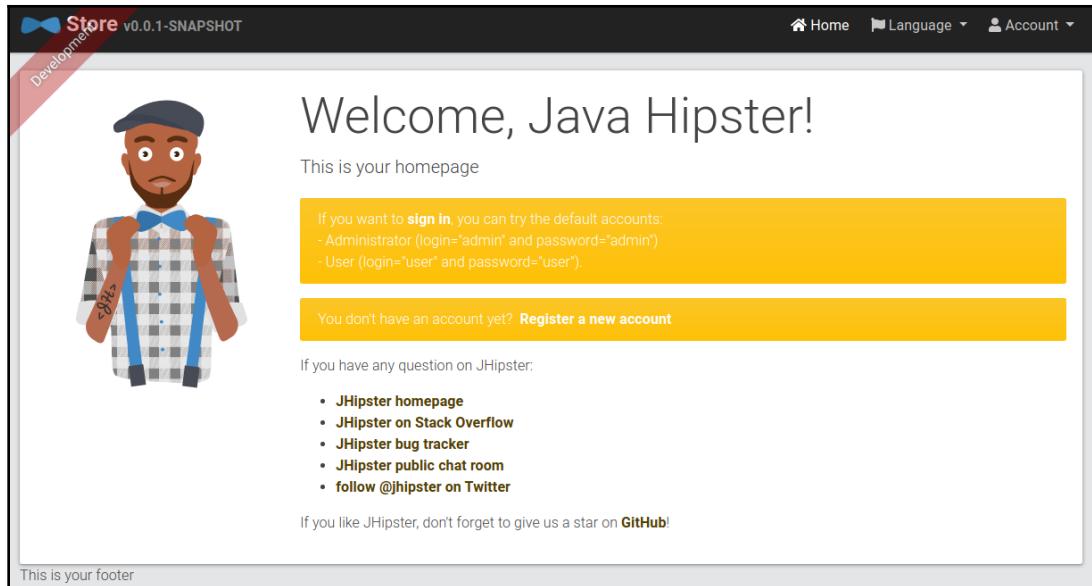
Now we are all set. We have a gateway application, we have a microservice with a SQL DB that runs with H2 in a `dev` profile and MySQL in a `prod` profile (invoice application), we have a microservice with MongoDB (notification application), and we just finished setting up our JHipster Registry locally. Now, it is time to start everything locally and see how seamless our microservice setup works. Make sure the registry is still running.

## Gateway application pages

Head over to the Terminal and go to the `e-commerce-app` folder. Navigate to the `store` folder and start the gateway application in `dev` mode:

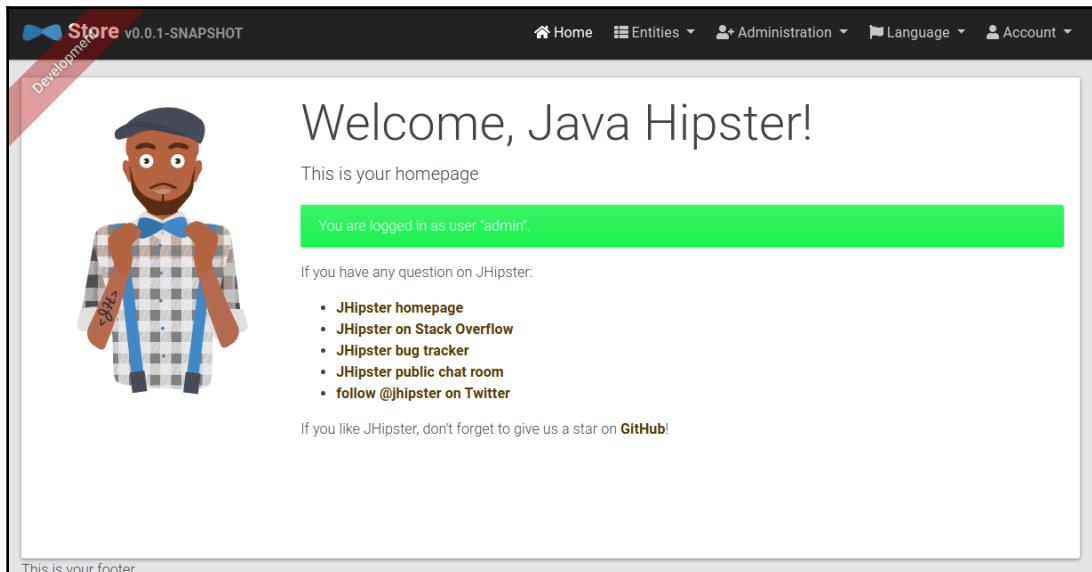
```
> cd store  
> ./gradlew
```

This will start our gateway application on port 8080. Let's open `http://localhost:8080` in our favorite browser:

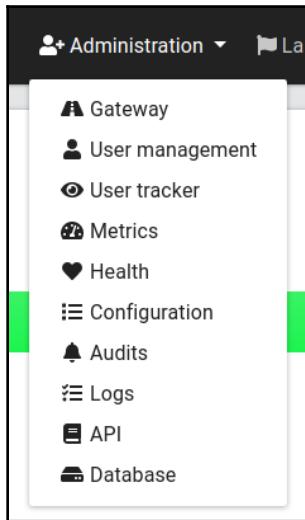


Now, we can click on the **Sign in** button on the home page or **Account | Sign in** from the top menu and enter the username and password as `admin` and `admin`, respectively.

Once you've logged in as an admin user, you will see the **Administration** menu:



In the **Administration** menu, you can find the following pages:



This includes the following:

**Gateway:** The **Gateway** page will show the list of microservice applications for which this application acts as a gateway. It will also show the routes and the services that handle the route, as well as the available servers for the route:

A screenshot of the "Gateway" page. The top navigation bar includes "Home", "Entities", "Administration", "Language", and "Account". The main content area is titled "Gateway" and "Current routes". It features a table with columns "URL", "Service", and "Available servers". A "REFRESH" button is in the top right. At the bottom, there is a footer message: "This is your footer".

Currently, no microservice application has been booted up, so the page is empty. We will see how this page is changed once we start our notification and invoice services soon.

**User management:** This is similar to monolithic user management and holds the basic user information and management options.

**Metrics:** The metrics page holds information about JVM metrics and service/DB statistics. This is, again, similar to the monolithic application. Added to that, this also shows the metric statistics for the microservice applications that have been registered.

**Health:** The health page shows the basic health information of the various services that we have in our application:

The screenshot shows the 'Health Checks' page of the JHipster Registry. At the top, there's a navigation bar with links for Home, Entities, Administration, Language, and Account. A red diagonal banner on the left says 'Development Store v0.0.1-SNAPSHOT'. Below the banner, the title 'Health Checks' is displayed, along with a 'REFRESH' button. The main content is a table with columns for Service name, Status, and Details. The table lists the following services and their statuses:

Service name	Status	Details
Disk space	UP	🔗
Database	UP	🔗
Microservice Refresh Scope	UP	🔗
Discovery Composite	UP	🔗
ClientConfigServer	UP	🔗
Hystrix	UP	🔗

At the bottom left, it says 'This is your footer'.

Similar to the monolithic application, it shows **Disk space** and **Database**. However, it also shows the health of the Discovery network (that is, the Discovery client and the Eureka server). It also shows the microservice config server's health, which is `spring-cloud-config-server`, and then shows the health of the circuit breaker we're using (`Hystrix`).

The **Configuration**, **Audits**, **Logs**, and **API** pages are similar to the ones that we saw for the monolithic application.

## JHipster Registry pages

Since we have started the registry server on port 8761, we can visit `http://localhost:8761` and log in with `admin` as the username and `admin` (the password that we provided when we started the application) as the password.

## Dashboard

Upon logging in, JHipster Registry shows the following information in the form of a dashboard:

The screenshot displays the JHipster Registry v5.0.2 dashboard with the following sections:

- System Status**: Shows environment (JHipster-Environment), data center (JHipster-DataCenter), current time (2019-12-08T15:59:35 +0000), system uptime (00:10), and below renew threshold (true).
- Instances Registered**: Shows a single instance (store) with ID store:5700a34b42d5b1694ee3eebbcd1794b7, marked as UP.
- General Info**: Shows total available memory (344mb), current memory usage (125mb (36%)), number of CPU (12), instance IP address (172.18.0.3), and instance status (UP).
- Health**: Monitors several services: DiskSpace (UP), RefreshScope (UP), DiscoveryComposite (UP), ConfigServer (UP), and Hystrix (UP).

JHipster Registry - [www.jhipster.tech](http://www.jhipster.tech)

These are grouped into sections, as follows.

### System status

This panel will show the environment in which the application is running and how long the application has been running (system uptime).

**Below renew threshold:** Our applications will have to send heartbeat signals to the registry service to notify the registry that the application is alive and running. The registry services rely on this heartbeat signal to register and deregister the application. That is, the existence of the application is determined with the help of this heartbeat ping. This is what will happen in the renew phase.

However, when the Eureka server is booting up, it will try to get all the information about instance registries from the nearby service. If the nearby service fails for any reason, then it will try to connect to all of its peers to fetch this information. If the Eureka server was able to fetch the information for all the servers, then it will set the renewal threshold based on the information received. Based on this information, JHipster Registry will hold the information on whether the current level is below the renewal threshold specified and notify users in the UI.

## Instances registered

This will show basic information about the instances that have been registered with the registry. Since we have only booted up the gateway service, we will see only one instance here. Basically, this will list all the instances that are connected to this registry service.

It shows the status of the system, the name of the system, and then the instance ID. The instance ID is generated based on the configuration in the `application.yml` file of JHipster Registry. It assigns a random value.

## General info and health

This also shows general information about the JHipster Registry service and health information of the cluster of services, similar to the gateway's health. The data here is fetched with the help of Spring Actuator's health and metric endpoints.

## Application listing page

This page lists the applications that are registered in the JHipster Registry service.

Navigate to **Administration | Gateway**:

The screenshot shows the 'Application Instances' section of the Registry interface. At the top, there's a header with the Registry logo and version v5.0.2, along with links for Home, Eureka, Configuration, Administration, and a refresh button. Below the header, the title 'Application Instances' is displayed, followed by a blue button labeled 'STORE' and a status indicator '1/1'. The main area is titled 'Instances' and contains a table with two columns: 'ID' and 'Status'. A single row is shown for the instance 'store:5700a34b42d5b1694ee3eebbcd1794b7', which has the status 'UP' and is associated with 'zone primary', 'profile dev', 'version 0.0.1-SNAPSHOT', 'management\_port 8080', 'git-commit 4309831', and 'git-branch master'.

This page shows the following information:

- The current instance ID and its name
- The current status of the instance
- The version that has been deployed
- The profile
- The zone in which it is deployed

The version number is fetched from the `build.gradle` or `pom.xml` file for Gradle and Maven projects, respectively.



The zone here normally refers to an Amazon zone. It is used by Ribbon to route the request to the nearest server. This configuration is useless if you don't use Amazon, and this is why we force it to **primary** (otherwise, the load balancing algorithm would be wrong).

All the pages in the administration module will have a drop-down menu that lists the various instances that are registered. From here, we can select an instance to view its metrics, health, configuration, and other information, depending on the page we are on.

## Metrics page

By default, this will show the registry's JVM metrics and its service statistics:

The screenshot shows the JHipster Registry application metrics interface. At the top, there's a navigation bar with links for Home, Eureka, Configuration, Administration, and a refresh button. Below the navigation is a search bar labeled "Search an application..." with the text "jhipster-registry". A dropdown menu is open, showing "JHIPSTER-REGISTRY" (selected), "Refresh now", and "disabled".

**Application Metrics**

**JVM Metrics**

**Memory**

- PS Eden Space (53M / 142M) Committed: 140M
- Code Cache (29M / 240M) Committed: 29M
- Compressed Class Space (9M / 1,024M) Committed: 10M
- PS Survivor Space (13M / 14M) Committed: 14M
- PS Old Gen (36M / 342M) Committed: 187M
- Metaspace 73M Committed: 76M

**Garbage collector statistics**

GC Live Data Size/GC Max Data Size (35M / 342M)	GC Memory Promoted/GC Memory Allocated (15M / 2,172M)	Classes loaded	15294				
Count	Mean	Min	p50	p75	p95	p99	Max
jvm.gc.pause	1	248	0	0	0	0	0

**Threads**

Runnable 16	Timed Waiting (34)	Waiting (99)	Blocked (0)
1%	32%	66%	

Total: 149 [Expand](#)

**System**

Uptime: 22 minutes 5 seconds  
Start time: Sunday, December 8, 2019 at 4:48:45 PM GMT+01:00

System CPU count	12
System 1m Load average	1.09
Process files max	1,048,576
Process files open	72

From here, we can select any instance from the drop-down menu provided and see its statistics, thus making JHipster Registry a single point of information that provides all the necessary insight into your microservice architecture:

The screenshot shows the JHipster Registry application metrics interface. The dropdown menu at the top is now set to "STORE (store:5700a34b42d5b1694ee3eebbcd1794b7)".

**Application Metrics**

**JVM Metrics**

**Memory**

- PS Eden Space (1,011M / 2,560M) Committed: 1,522M
- Code Cache (33M / 240M) Committed: 34M
- Compressed Class Space (16M / 1,024M) Committed: 17M
- PS Survivor Space (34M / 42M) Committed: 42M
- PS Old Gen (49M / 5,295M) Committed: 395M
- Metaspace 117M Committed: 123M

**Garbage collector statistics**

GC Live Data Size/GC Max Data Size (49M / 5,295M)	GC Memory Promoted/GC Memory Allocated (62M / 2,799M)	Classes loaded	25341				
Count	Mean	Min	p50	p75	p95	p99	Max
jvm.gc.pause	2	111	0	0	0	0	0

**Threads**

Runnable 24	Timed Waiting (19)	Waiting (73)	Blocked (0)
21%	16%	43%	

Total: 116 [Expand](#)

**System**

Uptime: 12 minutes 20 seconds  
Start time: Sunday, December 8, 2019 at 4:59:08 PM GMT+01:00

Process CPU usage	0.28 %
System CPU usage	11.79 %
System CPU count	12
System 1m Load average	1.59
Process files max	524,288
Process files open	338

For example, upon selecting the **Store** application instance, we will get store gateway-related information, as shown in the preceding screenshot.

## Health page

The health page will list the health of the registry itself and all the instances that are connected to it:

The screenshot shows the JHipster Registry Health Checks page. At the top, there is a search bar with the placeholder "Search an application..." and a dropdown menu set to "disabled". Below the search bar, a table lists service names and their statuses. The table has three columns: "Service Name", "Details", and a column for status indicators (green circles with "UP"). The services listed are DiskSpace, Db, RefreshScope, DiscoveryComposite, ClientConfigServer, and Hystrix. In the "Details" column, there is a link to "STORE (store:5700a34b42d5b1694ee3eebbcd1794b7)". A modal window is open over this link, showing detailed information about the selected store instance. The modal title is "STORE:5700a34b42d5b1694ee3eebbcd1794b7". It contains a search bar with the placeholder "Search an application...", a dropdown menu set to "disabled", and a "Details" section which is currently empty.

Service Name	Details	Status
DiskSpace		UP
Db		UP
RefreshScope	<a href="#">STORE (store:5700a34b42d5b1694ee3eebbcd1794b7)</a>	UP
DiscoveryComposite		UP
ClientConfigServer		UP
Hystrix		UP

For example, upon selecting the **Store** application instance, we will get store gateway-related information, as shown in the preceding screenshot.

## Configuration page

Similar to the health and metrics pages, JHipster Registry will provide detailed configuration information for all the instances connected to it. We can choose these instances from the drop-down menu:

The screenshot shows the JHipster Registry configuration interface. At the top, there's a navigation bar with links for Home, Eureka, Configuration, and Administration. Below the navigation is a search bar labeled "Search an application..." containing the text "jhipster-registry". A dropdown menu is open, showing "JHIPSTER-REGISTRY" with a green "UP" status indicator. To the right of the dropdown are two buttons: "Refresh now" and "disabled".

The main area is titled "Configuration" and has a sub-section "Spring configuration". It lists properties under "Prefix" and "Properties".

Prefix	Properties
application	
eureka.client	fetchRegistry gZipContent registryFetchIntervalSeconds initialInstanceInfoReplicationIntervalSeconds

On the right side of the configuration table, there are two columns of status indicators:

- Column 1: true, true
- Column 2: 10, 40

The preceding screenshot shows the configuration screen for the store application.

## Logs page

Similar to the preceding pages, the log page will also show the real-time logs of the application. This is really useful for debugging and getting more information when there is a failure:

The screenshot shows the JHipster Registry logs page. At the top, there's a navigation bar with links for Home, Eureka, Configuration, and Administration. Below the navigation is a search bar labeled "Search an application..." containing the text "JHIPSTER-REGISTRY". A dropdown menu is open, showing "JHIPSTER-REGISTRY" with a green "UP" status indicator. To the right of the dropdown are three buttons: "Refresh now", "disabled", and a dropdown menu.

The main area is titled "View Logs" and displays a scrollable log output for the "JHIPSTER-REGISTRY" application. The log entries are timestamped and show various DEBUG and INFO level messages related to the ZUULUpdaterService and ZuulUpdaterService.

```

io.github.jhipster.registry.service.ZuulUpdaterService.updateZuulRoutes() with arguments[] = []
2019-12-08 14:21:55.761 DEBUG 1 --- [scheduling-1] i.g.j.r.service.ZuulUpdaterService      : Checking instance store:a870b550b20bf3bba6292c5908449832 - Bottom
http://192.168.2.177:8080/
2019-12-08 14:21:55.761 DEBUG 1 --- [scheduling-1] i.g.j.r.service.ZuulUpdaterService      : Instance 'store:a870b550b20bf3bba6292c5908449832' already registered
2019-12-08 14:21:55.761 DEBUG 1 --- [scheduling-1] i.g.j.r.aop.logging.LoggingAspect      : Exit:
io.github.jhipster.registry.service.ZuulUpdaterService.updateZuulRoutes() with result = null
2019-12-08 14:22:00.761 DEBUG 1 --- [scheduling-1] i.g.j.r.aop.logging.LoggingAspect      : Enter:
io.github.jhipster.registry.service.ZuulUpdaterService.updateZuulRoutes() with arguments[] = []
2019-12-08 14:22:00.762 DEBUG 1 --- [scheduling-1] i.g.j.r.service.ZuulUpdaterService      : Checking instance store:a870b550b20bf3bba6292c5908449832 -
http://192.168.2.177:8080/
2019-12-08 14:22:00.762 DEBUG 1 --- [scheduling-1] i.g.j.r.service.ZuulUpdaterService      : Instance 'store:a870b550b20bf3bba6292c5908449832' already registered
2019-12-08 14:22:00.762 DEBUG 1 --- [scheduling-1] i.g.j.r.aop.logging.LoggingAspect      : Exit:
io.github.jhipster.registry.service.ZuulUpdaterService.updateZuulRoutes() with result = null
2019-12-08 14:22:05.762 DEBUG 1 --- [scheduling-1] i.g.j.r.aop.logging.LoggingAspect      : Enter:
io.github.jhipster.registry.service.ZuulUpdaterService.updateZuulRoutes() with arguments[] = []
2019-12-08 14:22:05.763 DEBUG 1 --- [scheduling-1] i.g.j.r.service.ZuulUpdaterService      : Checking instance store:a870b550b20bf3bba6292c5908449832 - Top
http://192.168.2.177:8080/
2019-12-08 14:22:05.763 DEBUG 1 --- [scheduling-1] i.g.j.r.service.ZuulUpdaterService      : Instance 'store:a870b550b20bf3bba6292c5908449832' already registered
2019-12-08 14:22:05.763 DEBUG 1 --- [scheduling-1] i.g.j.r.aop.logging.LoggingAspect      : Exit:
io.github.jhipster.registry.service.ZuulUpdaterService.updateZuulRoutes() with result = null
2019-12-08 14:22:10.763 DEBUG 1 --- [scheduling-1] i.g.j.r.aop.logging.LoggingAspect      : Enter:
io.github.jhipster.registry.service.ZuulUpdaterService.updateZuulRoutes() with arguments[] = []
2019-12-08 14:22:10.763 DEBUG 1 --- [scheduling-1] i.g.j.r.service.ZuulUpdaterService      : Checking instance store:a870b550b20bf3bba6292c5908449832 -
http://192.168.2.177:8080/
2019-12-08 14:22:10.763 DEBUG 1 --- [scheduling-1] i.g.j.r.service.ZuulUpdaterService      : Instance 'store:a870b550b20bf3bba6292c5908449832' already registered
  
```



The logs are formatted at the application level. The console in the preceding screenshot shows `tail -f` for consolidating logs.

The preceding screenshot shows the logs from the registry application.

## Loggers

We can also control the log levels of the registry and all the connected apps from the loggers' screen, which is as follows:

A screenshot of a web-based logger configuration interface. At the top, there's a header bar with the 'Registry v5.0.2' logo, the URL 'cdp.packtpub.com', and navigation links for 'Home', 'Eureka', 'Configuration', and 'Administration'. Below the header, the main title is 'Loggers'. A message says 'There are 1682 loggers.' To the right of the title are three buttons: 'STORE:5700A34B42D5B1694EE3EBCB1794B7', 'Refresh now', and 'disabled'. There's also a 'Filter' input field. The main content is a table with two columns: 'Name' and 'Level'. The 'Name' column lists logger names like 'LiquibaseSchemaResolver', 'ROOT', 'ch', 'ch.qos', and 'ch.qos.logback'. The 'Level' column shows sets of buttons for each logger, with colors indicating the current log level: blue for TRACE, green for DEBUG, cyan for INFO (which is highlighted), yellow for WARN, and dark grey for ERROR.

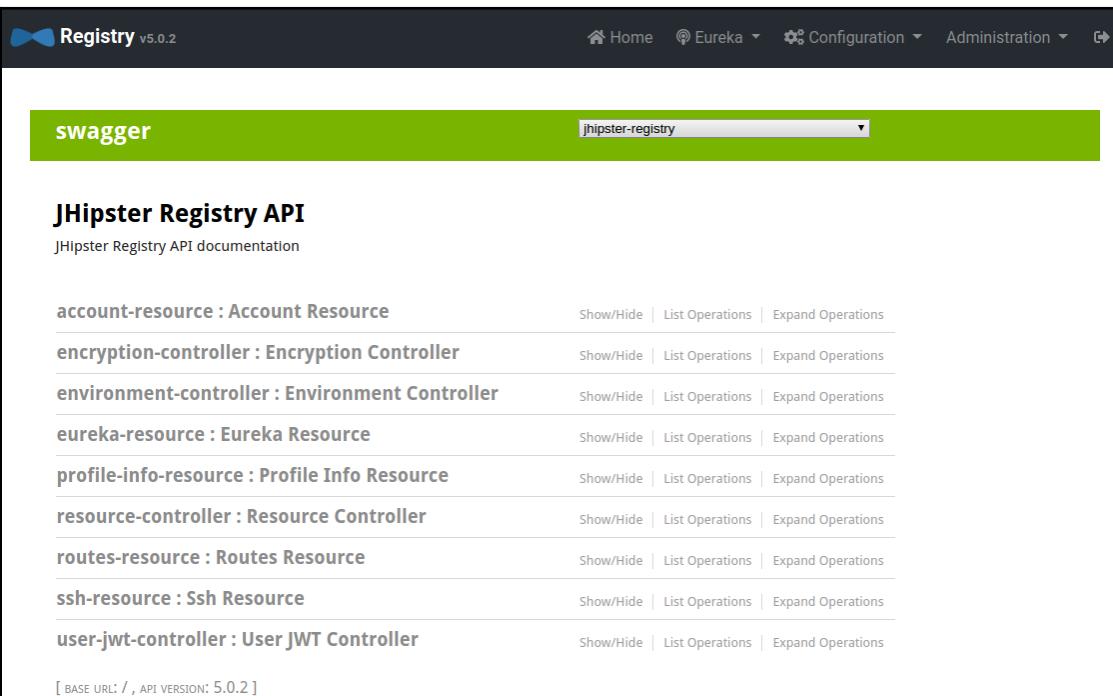
Name	Level
LiquibaseSchemaResolver	TRACE DEBUG INFO WARN ERROR
ROOT	TRACE DEBUG INFO WARN ERROR
ch	TRACE DEBUG INFO WARN ERROR
ch.qos	TRACE DEBUG INFO WARN ERROR
ch.qos.logback	TRACE DEBUG INFO WARN ERROR

The preceding screenshot shows the log-level config for the store application.

## Swagger API endpoints

The microservice architecture relies heavily on API calls between the gateway and services, services and the registry, and the gateway and registries. Therefore, it is essential for developers and users to get to know the API endpoints that they can access, as well as the information that's required to access those endpoints.

This can be a lot of work. Fortunately, libraries such as Swagger come to the rescue. We just have to add the standard comments to the methods; then, the Swagger API will do the necessary work to extract information from them and convert them into a beautiful user interface:



The screenshot shows the Swagger UI interface for the JHipster Registry API. At the top, there's a navigation bar with links for Home, Eureka, Configuration, Administration, and a back arrow. Below the navigation is a search bar with the text "jhipster-registry". The main content area has a green header bar with the word "swagger". The page title is "JHipster Registry API" and the subtitle is "JHipster Registry API documentation". The main body lists various API endpoints with their descriptions and operation options (Show/Hide, List Operations, Expand Operations). The listed endpoints include:

Endpoint	Description	Show/Hide	List Operations	Expand Operations
account-resource : Account Resource				
encryption-controller : Encryption Controller				
environment-controller : Environment Controller				
eureka-resource : Eureka Resource				
profile-info-resource : Profile Info Resource				
resource-controller : Resource Controller				
routes-resource : Routes Resource				
ssh-resource : Ssh Resource				
user-jwt-controller : User JWT Controller				

At the bottom left, there's a note: "[ BASE URL: / , API VERSION: 5.0.2 ]".

The preceding screenshot shows the default generated Swagger UI page. It lists all the endpoints that are available and then supplies the list of operations that it provides. It shows the playground where we can frame requests and test them for output.



Normally, the Swagger API docs are only available in development mode. If you are developing an API service and if there is a need to expose this to end users or external developers using your service, you can enable it in production by setting the `swagger` profile, along with the `prod` profile, by setting the `spring.profiles.active=prod,swagger` property in the `application-prod.yaml` file.

Similar to the other pages, this also lists the various instances that are connected to this registry service. We can select them from the drop-down menu (upper-right corner) to see what APIs are provided by various applications:

The screenshot shows the Registry v5.0.2 interface with the title "store API". Below it, the sub-section "store API documentation" is visible. A green header bar at the top contains the word "swagger". To its right is a dropdown menu with the value "store:5700a34b42d5b1694ee3eebbcd1794b7". The main content area lists various resource types with their corresponding operations:

Resource Type	Action	Show/Hide	List Operations	Expand Operations
account-resource : Account Resource		Show/Hide	List Operations	Expand Operations
customer-resource : Customer Resource		Show/Hide	List Operations	Expand Operations
gateway-resource : Gateway Resource		Show/Hide	List Operations	Expand Operations
order-item-resource : Order Item Resource		Show/Hide	List Operations	Expand Operations
product-category-resource : Product Category Resource		Show/Hide	List Operations	Expand Operations
product-order-resource : Product Order Resource		Show/Hide	List Operations	Expand Operations
product-resource : Product Resource		Show/Hide	List Operations	Expand Operations
user-jwt-controller : User JWT Controller		Show/Hide	List Operations	Expand Operations
user-resource : User Resource		Show/Hide	List Operations	Expand Operations

The operations listed in the gateway API will provide the following information:

The screenshot shows the "store API" documentation for the "account-resource : Account Resource". It lists several operations:

Method	Path	Description
GET	/api/account	getAccount
POST	/api/account	saveAccount
POST	/api/account/change-password	changePassword
POST	/api/account/reset-password/finish	finishPasswordReset
POST	/api/account/reset-password/init	requestPasswordReset
GET	/api/activate	activateAccount
GET	/api/authenticate	isAuthenticated
POST	/api/register	registerAccount

Below this, there is another section for "customer-resource : Customer Resource" with similar options.

This lists all the operations that are available in the AccountResource file. It shows the method type (GET/POST/PUT/DELETE), as well as the endpoint and the method name that is present in the AccountResource file:

The screenshot shows the store API documentation for the account-resource module. At the top, it says "store API" and "store API documentation". Below that, it shows a specific operation: "account-resource : Account Resource". The method is "GET" and the endpoint is "/api/account". To the right of the endpoint, there are three buttons: "getAccount", "Show/Hide", and "List Operations | Expand Operations". The main content area displays the "Response Class (Status 200)" which is "OK". There are two tabs: "Model" and "Example Value", with "Model" currently selected. It shows a JSON object structure:

```
{  
    "activated": true,  
    "authorities": [  
        "string"  
    ],  
    "createdBy": "string",  
    "createdDate": "2019-12-08T16:26:12.084Z",  
    "email": "string",  
    "firstName": "string",  
    "id": 0,  
    "lastName": "string",  
    "password": "string",  
    "status": "string",  
    "username": "string",  
    "version": 0  
}
```

Below this, there is a "Response Content Type" dropdown set to "application/json". Under "Response Messages", there is a table with columns: "HTTP Status Code", "Reason", "Response Model", and "Headers". It lists three rows: 401 (Unauthorized), 403 (Forbidden), and 404 (Not Found). A "Try it out!" button is located below the table. At the bottom, there is another row for a POST request to "/api/account" with a "saveAccount" button.

Upon clicking any one of the endpoints that's available, you will see detailed information about the response classes, response errors, response content type, as well as how the response object is structured. In addition to this, it also shows how the model object is constructed. These are particularly helpful for end users who want to access these APIs:

```
UserDTO {  
    activated (boolean, optional),  
    authorities (Array[string], optional),  
    ...  
}
```

```
        createdBy (string, optional),  
        createdDate (string, optional),  
        email (string, optional),  
        firstName (string, optional),  
        id (integer, optional),  
        imageUrl (string, optional),  
        langKey (string, optional),  
        lastModifiedDate (string, optional),  
        lastModifiedBy (string optional),  
        lastName (string, optional),  
        login (string)  
    }
```

Next, there is the option to try out the endpoint by clicking the **Try it out** button:

The screenshot shows a Swagger UI interface with the following sections:

- Curl**: A code editor containing a curl command to make a GET request with headers Accept: application/json and Authorization: Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhZG1pbk... .
- Request URL**: A text input field containing the URL `http://localhost:8761/services/store/store:5700a34b42d5b1694ee3eebbcd1794b7/api/account`.
- Response Body**: A JSON response object displayed in a code editor:

```
{  
  "id": 3,  
  "login": "admin",  
  "firstName": "Administrator",  
  "lastName": "Administrator",  
  "email": "admin@localhost",  
  "imageUrl": "",  
  "activated": true,  
  "langKey": "en",  
  "createdBy": "system",  
  "createdDate": null,  
  "lastModifiedBy": "system",  
  "lastModifiedDate": null,  
  "authorities": [  
    "ROLE_USER",  
    "ROLE_ADMIN"  
  ]  
}
```
- Response Code**: A text input field containing the value 200.

This shows the request and its response. It also shows how to frame the request, along with the authentication token:

Also, it provides the response code and the response header information that is returned by the server, which is also extremely useful for API programmers.

# Running invoice and notification applications locally

We have started the store gateway and registry services. Now, we can go to our invoice and notification application folders and run them locally:

```
> cd invoice  
> ./gradlew
```

This will start the service on port 8081.

Open another Terminal and run the following command:

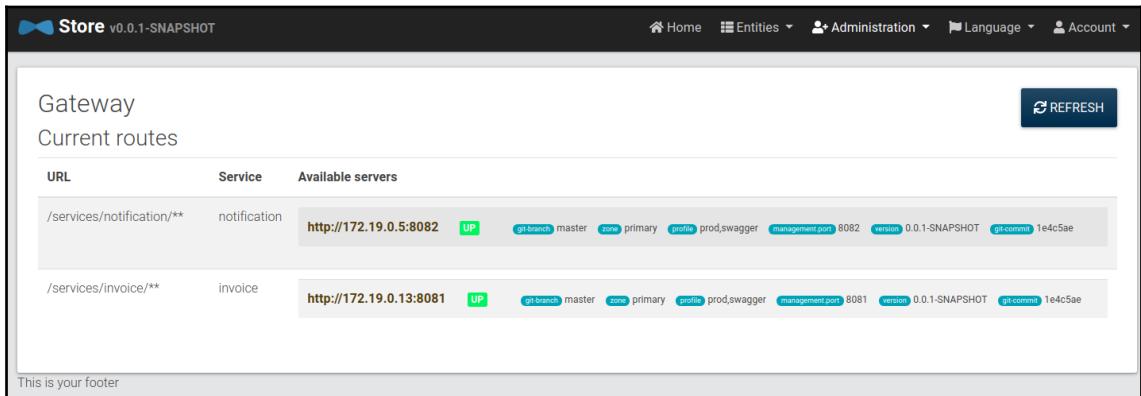
```
> cd notification  
> docker-compose -f src/main/docker/mongodb.yml up -d  
> ./gradlew
```

This will start the MongoDB server with Docker and run the microservice on port 8082:

```
-----  
Config Server: Connected to the JHipster Registry running in Docker  
-----  
<=====--> 83% EXECUTING [4m 21s]
```

Upon starting the application, it will also try to connect to JHipster Registry and register itself. Watch out for the preceding message once your server has started to make sure that it is connected to JHipster Registry.

Alternatively, you can test this via your gateway application. Log in to your gateway application and navigate to **Administration | Gateway**:



The screenshot shows the 'Gateway' section of the JHipster Registry interface. It lists two services: 'notification' and 'invoice'. The 'notification' service is running at <http://172.19.0.5:8082> and the 'invoice' service is running at <http://172.19.0.13:8081>. Both services are marked as 'UP'. The table includes columns for URL, Service, and Available servers. A 'REFRESH' button is located in the top right corner. At the bottom left, there is a footer note: 'This is your footer'.

Here, you can see the two microservice applications (**INVOICE** and **NOTIFICATION**) that have been booted up. These are available at their respective URLs.

You can also check the JHipster Registry home page to see the registered instances:

Instances Registered		
App	Instance ID	Status
INVOICE	invoice:0586b50366f6bb8fc51726c97c28a1df	UP
NOTIFICATION	notification:c0d97497619b384cb9953074316858d9	UP
STORE	store:5700a34b42d5b1694ee3eebbcd1794b7	UP

Similarly, all the other pages in JHipster Registry will start to show invoice or notification as one of the instances. Using this, we can get their health, configuration, logs, and metrics direct from JHipster Registry:

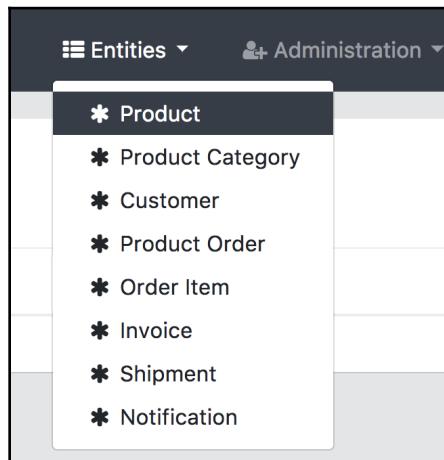
```
└── app.jdl
    ├── invoice
    ├── notification
    └── store
```

If you have followed along with this book, the preceding code will be the directory structure you will have in the e-commerce-app folder.

## Explaining the generated entity pages

Once the application has started successfully, you will see that the entities are generated in the gateway application and are available under the entity **nav** menu.

This includes all the gateway entities, as well as the microservice entities:



The following is the invoice screen that's created in the gateway application:

### Create or edit a Invoice

Code

Date  
 dd/mm/yyyy, --:-

Details

Status

Payment Method

Payment Date  
 dd/mm/yyyy, --:-

Payment Amount

Product Order Id

CANCEL  SAVE

Try to create a few entities to verify that everything is working fine.

## Summary

In this chapter, we have successfully generated a gateway and two microservices. We downloaded JHipster Registry and started it locally. Then, we successfully segregated and generated the entity files for a notification and invoice service. After that, we booted up all our applications and saw the generated code and explored the application screens. By doing this, we learned about using the JHipster Registry and setting it up locally. Last but not least, we committed all our changes to Git (in other words, reached a checkpoint).

In the next chapter, we will learn how to deploy our microservice stack using Docker and Docker Compose. We will also learn about the different deployment options provided by JHipster for microservices.

# 5

## Section 5: Deployment of Microservices

This section first introduces you to advanced local and cloud deployment options for microservices with the help of Docker Compose and JHipster. You will understand local deployment and testing techniques through the Google Cloud deployment of the generated microservice stack using Kubernetes and JHipster.

This section comprises the following chapters:

- Chapter 11, *Deploying with Docker Compose*
- Chapter 12, *Deploying to the Cloud with Kubernetes*

# 11

# Deploying with Docker Compose

In the previous chapter, we generated the microservice application stack. Now, it's ready for production. In this chapter, we will focus on how to deploy the application using Docker Compose. We will also look at the various options that JHipster provides for deployment, followed by how to deploy JHipster Registry and JHipster Console alongside the application.

In this chapter, we will cover the following topics:

- Introducing microservice deployment options
- Generated Docker Compose files
- Generating Docker Compose files for microservices

By doing this, we will learn how to deploy our stack locally with JHipster Registry and JHipster Console using Docker Compose.

## Introducing microservice deployment options

*"The success of an application not only depends on how well we design it. It depends on how well we implement, deploy, and maintain it."*

A well-designed microservice application in a low-availability environment is useless. Therefore, it is equally important to decide on a deployment strategy that increases its chances to succeed. When it comes to deployment, there are a plethora of tools available. Each one of them has its pros and cons, and we have to choose one that is suitable for our needs. JHipster currently provides sub-generators so that we can create configuration files to containerize, deploy, and manage the microservices via the following methods:

- Docker and Docker Compose
- Kubernetes (also helps to orchestrate your deployment)
- OpenShift (also provides private cloud deployment and orchestration)

We will look at these in detail in the following sections.

## A short introduction to Docker Compose

Before looking at what Docker Compose is, let's go through some facts. Shipping code to the server is always difficult, especially when you want to scale it. This is mainly because we have to manually create the same environment and make sure the application has all the necessary connectivity (to other services). This was a major pain point for teams while shipping and scaling their code.

*"Shipping code to the server is difficult."*

Containers were the game-changer in this field. They helped bundle the entire application along with dependencies in a shippable container, and all we need is to provide an environment in which these containers can run. This simplified the process of shipping code to the server among development teams. This also reduced the amount of time a team spent making sure that the application ran seamlessly across the environment.

Containers solve the application deployment problem, but how do we scale them?

Docker Compose is an option we can consider. First, let's see what Docker Compose is, and then see what problems it solves.

Docker Compose is a tool that helps us define and run multi-container Docker applications with a single file. That is, we use a `.yaml` file to define the requirements and/or dependencies of the application. Then, with Docker Compose, we can create newer deployments and start our applications, as defined in the Docker Compose file. We can also use Docker Compose to seamlessly scale the deployed applications.

So, what is required in a Docker Compose file?

The following code segment is a sample Docker Compose file that will start a Redis database on port 5000:

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
  redis:
    image: "redis:alpine"
```

Let's take a look at the code segment sequentially:

- The first line of the Docker Compose file should be the version of the Docker Compose tool.
- Then, we need to specify all the necessary services that we need for our application to run. They should be defined in the `services:` section.
- We can also define multiple services here, giving a name to each (`web` and `redis`).
- This is followed by how to build the `service` (either via a command to build or referring to a Docker image).
- If the application needs any port access, we can configure it using `5000:5000` (that is, internal port: external port).
- Then, we have to specify the volume information. This basically tells Docker Compose to serve the files from the location specified.

Once we have specified the services that are required for our application, we can start the application via Docker Compose. This will start the entire application, along with the services, and expose the services on the port specified.

With Docker Compose, we can perform the following operations:

- **Start:** `docker-compose -f <docker_file> up`
- **Stop:** `docker-compose -f <docker_file> down`

We can also perform the following operations:

- **List the running services and their status:** `docker ps`
- **Logs:** `docker log <container_id>`

In the Compose file, we can add the project name, as follows:

```
version: '3'  
COMPOSE_PROJECT_NAME: "myapp"  
services:  
  web:  
    build: .  
    ports:  
      - "5000:5000"  
  redis:  
    image: "redis:alpine"
```

This can be used to identify multiple environments. With the help of this, we can isolate multiple environments. This helps us handle multiple instances across various dev, QA, and prod environments.

Docker Compose itself is a great tool for deploying your application, along with all the services it needs. It provides infrastructure as a code. It is an excellent choice for development, QA, and other environments, except for production. But why?

Docker Compose is really good for creating and starting your application. However, when you want to update an existing container, there will be a definite downtime since Docker Compose will recreate the entire container (there are a few workarounds to make this happen but still, Docker Compose needs some improvement in this regard). In real-world scenarios, you would often need some kind of orchestration and advanced features for scalability.

## Introduction to Kubernetes

According to the Kubernetes website:

*"Kubernetes (K8s) is an open source system for automating the deployment, scaling, and management of containerized applications."*

It is a simple and powerful tool for managing containerized applications. It provides zero downtime when you roll out a newer application or update an existing application. You can automate it to scale in and out based on certain factors. It also provides self-healing so that Kubernetes automatically detects the failing application and spins up a new instance. We can also define secrets and configuration that can be used across instances.



Kubernetes is kind of the *de facto* standard when it comes to container orchestration, the same way Docker is the *de facto* standard for creating containers.

Kubernetes primarily focuses on zero downtime production application upgrades and also scales them as required.

A single deployable component is called a **pod** in Kubernetes. This can be as simple as a running process in the container. A group of pods can be combined together to form a **deployment**.

Similar to Docker Compose, we can define the applications and their required services in a single YAML file or multiple files (as per our convenience).

The following code is a sample Kubernetes file that will start an NGINX server:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

Here, we start with `apiVersion` in a Kubernetes deployment file. This is followed by the type, which takes a pod, deployment, service, namespace, Ingress (load balancing the pods), role, or something else.



Ingress forms a layer between the services and the internet so that all the inbound connections are controlled or configured with the Ingress controller before they are sent to Kubernetes services on the cluster. On the other hand, the egress controller controls or configures services that are going out of the Kubernetes cluster.

This is followed by the `metadata` information, such as the type of environments, the deployment name (`nginx-deployment`), and labels (Kubernetes uses this information to identify and segregate the pods). Kubernetes uses this metadata information to identify particular pods or a group of pods, and we can manage these instances with this metadata. This is one of the key differences with Docker Compose, where Docker Compose doesn't have the flexibility of defining the metadata pertaining to the containers.

This is followed by the `spec`, where we define the replicas, selectors, and deployment template.

Within the `template` is `containers`, where the specification of the images or our application is defined. There can be one or more containers in a deployment. We can also define the pull strategy for our images, as well as the environment variables and their exposed ports. We can define the resource limitations on the machine (or VM) for a particular service. We can also define health checks here, that is, each service is monitored for its health and, when some services fail, they are immediately replaced by newer ones.

They also provide service discovery out of the box by assigning each pod an IP, which makes it easier for the services to identify and interact with them. They also provide a better dashboard so that you can visualize your architecture and the status of the application. You can do most of this management via this dashboard, including checking the status, logs, scaling the services up or down, and so on.

Since Kubernetes provides a complete orchestration of our services and deployments with configurable options, it makes it a bit hard to set up initially, and this means it is not ideal for a development environment. We also need the `kubectl` CLI tool for management.

Despite the fact that we use Docker images inside, the Docker CLI can't be used.

There is also **Minikube** (minified Kubernetes), which is used for developing and testing applications locally. Kubernetes can also be enabled on Docker Desktop on Mac and Windows for development and testing purposes.



Kubernetes not only takes care of containerizing your application, but it also helps you scale, manage, and deploy your application. It orchestrates your entire application deployment. Additionally, it provides service discovery, automated health checks, and many more features.

We will focus more on the Kubernetes sub-generator in the next chapter.

## Introduction to OpenShift

OpenShift is a multi-cloud, open source container application platform. It is based on Kubernetes and used for developing, deploying, and managing applications. It is a common platform for developers and operations. It helps them build, deploy, and manage applications consistently across hybrid cloud and multi-cloud infrastructures.

For developers, it provides a self-service platform in which they can provision, build, and deploy applications and their components. With automated workflows for converting your source into an image, it helps developers go from source to ready-to-run Dockerized images.

For operations, it provides a secure, enterprise-grade Kubernetes for policy-based controls and automation for application management, such as cluster services, scheduling, and orchestration with load balancing and autoscaling capabilities.

JHipster also provides OpenShift deployment files as a separate sub-generator. We can generate them by running `jhipster openshift` and answering the questions as needed. This will generate OpenShift-related deployment files. If you are familiar with OpenShift, try it out for the application stack we built.

## Generated Docker Compose files

By default, JHipster will generate Docker Compose files that allow us to run the application completely in the containerized environment, irrespective of the options chosen. For example, in the gateway application that we have generated, the following files are generated by default under `src/main/docker`, among a few others:

- `sonar.yml`: This file creates and starts a SonarQube server.
- `mysql.yml`: This file creates and starts a MySQL database server and creates a user and schema. If we choose another database, the Docker Compose file that was generated would also correspond to that.
- `monitoring.yml`: This file creates and starts Prometheus and Grafana for monitoring. Required configurations for the Grafana dashboard and Prometheus are also generated.

- `jhipster-registry.yml`: This file creates and starts a JHipster Registry service.
- `app.yml`: This is the main file that creates and starts the application, along with services such as JHipster Registry and the database.

In addition to this, JHipster also creates a JIB configuration, as we saw earlier, which helps you containerize the application.

Then, we will see a folder called `central-server-config`. This will be used as a central configuration server for the JHipster Registry.

When the registry and the application are running in Docker, it uses `application.yml` from the `docker-config` folder as the central configuration server.

On the other hand, when running only the registry in Docker mode, the application, not in Docker, will use `application.yml` from the `localhost-config` folder. The key difference is that the URL defining the Eureka client varies.

Now, let's look at the important Docker files that have been generated.

## Walking through the generated files

Let's start with the `app.yml` file under `src/main/docker`, which can be found inside your gateway application.

As we saw at the beginning of this chapter, the file starts with the Docker version that it supports:

```
version: '2'
```

This is followed by the services section, where various services, applications, or components that we will kick start with this Docker Compose file are defined.

Under the services section, we have a name for the service – in our case, `store-app` – followed by the image that is used as the container. This image is generated with the help of JIB from our Gradle build.

This is followed by a series of environment variables that our application will depend on, including the following:

- `SPRING_PROFILES_ACTIVE`: This tells the application to run in production mode and expose Swagger endpoints.

- EUREKA\_CLIENT\_SERVICE\_URL\_DEFAULTZONE: This tells the application where to check for the JHipster Registry (which is the Eureka client that we are using. If we have chosen Consul here, then the application will point to the Consul URL).
- SPRING\_CLOUD\_CONFIG\_URI: This tells the application where to look for the config service for the application.
- SPRING\_DATASOURCE\_URL: This tells the application where to look for the data source.
- JHIPSTER\_SLEEP: This is the custom property that we have used to make sure that the JHipster Registry starts before the application starts up.

Finally, the port that the application should run and be exposed on is specified:

```
store-app:  
  image: store  
  environment:  
    - _JAVA_OPTIONS=-Xmx512m -Xms256m  
    - SPRING_PROFILES_ACTIVE=prod,swagger  
    - MANAGEMENT_METRICS_EXPORT_PROMETHEUS_ENABLED=true  
    - EUREKA_CLIENT_SERVICE_URL_DEFAULTZONE=...  
    - SPRING_CLOUD_CONFIG_URI=...  
    - SPRING_DATASOURCE_URL=...  
    - JHIPSTER_SLEEP=30 # gives time for other services to boot  
  before the application  
  ports:  
    - 8080:8080
```

We have seen how the service is defined with the Docker Compose file. Now, let's look at two other services that are needed for our application to run. These are the database and JHipster Registry.

So, we have another service called `store-mysql`, which creates and starts the MySQL server. Since we already have MySQL as a separate Docker Compose file, it is linked here with an `extends` keyword, followed by the Docker Compose file and the service that we have to start from the specified file:

```
store-mysql:  
  extends:  
    file: mysql.yml  
    service: store-mysql
```

The compose file, `mysql.yml`, contains the following code:

```
version: '2'  
services:  
  store-mysql:
```

```
image: mysql:8.0.18
# volumes:
# - ~/volumes/jhipster/store/mysql/:/var/lib/mysql/
environment:
- MYSQL_USER=root
- MYSQL_ALLOW_EMPTY_PASSWORD=yes
- MYSQL_DATABASE=store
ports:
- 3306:3306
command: mysqld --lower_case_table_names=1 --skip-ssl --
character_set_server=utf8mb4 --explicit_defaults_for_timestamp
```

This started with the version of Docker Compose, followed by the `services` keyword, and then specified the service name, `store-mysql`, that is used in the `app.yml` file.



If you want to specify a volume for the persistent data storage, you can uncomment the commented `volumes` segment. This basically maps the local file location to Docker's internal location so that the data is persistent, even if the Docker image itself is replaced or updated. This is recommended.

This is followed by a set of environment variables, such as the username and the password (it is set to empty here, but for a real production application, it is recommended to set a more complex password), and then the database schema name.

The command that needs to run in order to start the MySQL server is defined next. Now, we need to go back to the `app.yml` file and look at the JHipster Registry service. This is again extending the `jhipster-registry.yml` file. One more thing to note here is that even though we extend the services from another Docker Compose file, we can override the environment variables that we specified in the original Docker Compose file. This comes in handy in certain cases where we have to kickstart our application with different or customized values. In this case, the Spring Cloud Config server file location is overridden:

```
jhipster-registry:
extends:
  file: jhipster-registry.yml
  service: jhipster-registry
environment:
- SPRING_CLOUD_CONFIG_SERVER_COMPOSITE_0_TYPE=native
-
SPRING_CLOUD_CONFIG_SERVER_COMPOSITE_0_SEARCH_LOCATIONS=file:
./central-config/docker-config/
```

The Jhipster-registry.yml file is as follows:

```
version: '2'
services:
  jhipster-registry:
    image: jhipster/jhipster-registry:v5.0.2
    volumes:
      - ./central-server-config:/central-config
    environment:
      - _JAVA_OPTIONS=-Xmx512m -Xms256m
      - SPRING_PROFILES_ACTIVE=dev,swagger
      - SPRING_SECURITY_USER_PASSWORD=admin
      - JHIPSTER_REGISTRY_PASSWORD=admin
      - SPRING_CLOUD_CONFIG_SERVER_COMPOSITE_0_TYPE=native
      -
      SPRING_CLOUD_CONFIG_SERVER_COMPOSITE_0_SEARCH_LOCATIONS=file:
        ./central-config/localhost-config/
        # - SPRING_CLOUD_CONFIG_SERVER_COMPOSITE_0_TYPE=git
        #
      SPRING_CLOUD_CONFIG_SERVER_COMPOSITE_0_URI=https://github.com/
        jhipster/jhipster-registry/
        #
    SPRING_CLOUD_CONFIG_SERVER_COMPOSITE_0_SEARCH_PATHS=central-config
  ports:
    - 8761:8761
```

The central-config file for JHipster Registry can be seen in the following code block. The JWT secret and Eureka client's URL are configured here. The JWT token that's specified allows services to authorize and communicate between themselves and the registry:

```
# Common configuration shared between all applications
configserver:
  name: Docker JHipster Registry
  status: Connected to the JHipster Registry running in Docker
jhipster:
  security:
    authentication:
      jwt:
        secret: my-secret-token-to-change-in-production
eureka:
  client:
    service-url:
      defaultZone: http://admin:${jhipster.registry.
        password}@localhost:8761/eureka/
```

Apart from these, JHipster also generated the `sonar.yml`, `hazelcast-management-center.yml`, and `monitoring.yml` files (these files are not important for deploying your application).

Similarly, in the microservices, that is, in our invoice and the notification applications, similar files will be generated. They are the same except for the change in the service name.

MongoDB can also run as a cluster with different nodes and configurations and, hence, it has a slightly more complex configuration here. Here, we have two Docker Compose files:

- `mongodb.yml` is for starting MongoDB with a single node.
- `mongodb-cluster.yml` is used to start MongoDB as a cluster.

Please check the database port number between the gateway and the microservice application that contains the MySQL database. Since they use the same database, there may be a clash in the port number since JHipster generates the same port number for both. Change it to any other unused port; otherwise, Docker Compose will throw an error. In our case, I have changed the exposed port mapping to `3307:3306` in the invoice service's `mysql.yml` file. This port is only used if you want to connect to the DB from outside the docker network; the invoice application can still access it via port `3306`, which is internal to the service.

## Building and deploying everything to Docker locally

There are multiple ways in which we can use the Docker Compose files based on our needs.

In general, when we are developing the application, we can run the application with the Maven or Gradle command so that we can debug the application and reload the changes faster, as well as start the database and JHipster registry with Docker.

Otherwise, you can start the entire application from the `app.yml` file, which will kickstart the database, JHipster Registry, and then the application itself. To do that, open your Terminal or Command Prompt, go to each of the application folders, and run the following commands.

First, Dockerize the application by taking a production build of our application with the following command:

```
> ./gradlew bootJar -Pprod jibDockerBuild
```

Once done, we can start the app via the `docker-compose up -d` command:

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

`-f` specifies the file that `docker-compose` should start the containers with. The `-d` flag tells `docker-compose` to run everything in detached mode. This will start the application in Docker and expose the application on ports 8080, 8081, and 8082, the registry server on port 8761, and the database on ports 3306, 3307, and 27017.

Once you have done this for all three applications, we can check the running Docker containers with the following command:

```
> docker ps -a
```

It should list all seven containers:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
delbaef0b470	jhipster/jhipster-registry:v5.0.2	"#!/bin/sh -c 'java -jar /opt/jhipster-registry.jar'"	2 minutes ago	Up 2 minutes	0.0.0.0:8761->8761/tcp 8082/tcp	docker_jhipster-registry_1
led9115906ca	notification	"bash -c 'chmod +x ./run.sh'"	2 minutes ago	Up 2 minutes	33060/tcp, 0.0.0.0:3307->3306/tcp	docker_notification-app_1
473aae09a92b	mysql:8.0.18	"docker-entrypoint.su..."	2 minutes ago	Up 2 minutes	33060/tcp	docker_invoice-mysql_1
332313333333	invoice	"bash -c 'chmod +x ./run.sh'"	2 minutes ago	Exited (1) About a minute ago	0.0.0.0:27017->27017/tcp 0.0.0.0:3306->3306/tcp, 33060/tcp	docker_invoice-app_1
697ae0653e9e	store	"bash -c 'chmod +x ./run.sh'"	2 minutes ago	Exited (1) 2 minutes ago	0.0.0.0:27017->27017/tcp	docker_store-app_1
0e36a9fcf5a2	mongo:4.0.13	"docker-entrypoint.su..."	25 hours ago	Up 2 minutes	0.0.0.0:33060->3306/tcp	docker_notification-mongodb_1
3b0ea011df68	mysql:8.0.18	"docker-entrypoint.su..."	44 hours ago	Up 43 hours	33060/tcp	docker_store-mysql_1

As you can see, there are three app containers (**store**, **notification**, and **invoice**), as well as a JHipster Registry, followed by three database containers (two MySQL and one MongoDB). Their order may vary).

You can view the logs from the containers by running `docker logs <container id>`. You will see the application running on `http://localhost:8080`, and the registry running on `http://localhost:8761`.



You can shut down these instances by running `docker-compose -f src/main/docker/app.yml down` on each of the application folders or by running `docker kill <container id>` for each of the preceding containers.

Now, let's learn how we can do the same with the `docker-compose` sub-generator.

# Generating Docker Compose files for microservices

There are many Docker Compose files and maintaining them is hard. Thankfully, JHipster has a docker-compose sub-generator bundled with it. The docker-compose sub-generator helps you organize all your application's Docker Compose files together. It creates a single Docker Compose file that refers to all the applications and their database, along with the registry and monitoring.

Let's go to the base folder, create a folder, and name it `docker-compose`:

```
> mkdir docker-compose && cd docker-compose
```

Once inside the `docker-compose` folder, we can run the following command:

```
jhipster docker-compose
```

This will generate the Dockerfiles.

As usual, it will ask us a series of questions before generating the files:

```
> jhipster docker-compose
INFO! Using JHipster version installed globally
INFO! Executing jhipster:docker-compose
INFO! Options: from-cli: true
✓ Docker is installed
? Which *type* of application would you like to deploy?
  Monolithic application
  > Microservice application
```

First, it asks us which type of application we would like to deploy. We will select the microservice application option.

This is followed by choosing the type of gateway that we would like to use; there are two options available: a JHipster gateway with Zuul proxy, and the more exciting Traefik gateway with Consul.

Let's choose the JHipster gateway with Zuul proxy:

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Which *type* of gateway would you like to use? (Use arrow keys)
❯ JHipster gateway based on Netflix Zuul
Traefik gateway (only works with Consul)
```

Then, we have to select the location of the microservice gateway and applications. This is the main reason why we have generated the applications inside a single parent folder. This will help any plugins and sub-generators easily find the Docker configuration files we've created. We will select the default option (shown as `(..)` in the following screenshot):

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Which *type* of gateway would you like to use? JHipster gateway based on Netflix Zuul
? Enter the root directory where your gateway(s) and microservices are located (../) _
```

After selecting the location, JHipster will search inside the given folder for any application that was generated by JHipster and list them in the next question. In our case, it lists `notification`, `invoice`, and `store`. We can choose all of them (by pressing the spacebar key) and hit *Enter*:

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Which *type* of gateway would you like to use? JHipster gateway based on Netflix Zuul
? Enter the root directory where your gateway(s) and microservices are located ../
3 applications found at /home/deepu/Documents/jhipster-book/v2/e-commerce-app/
? Which applications do you want to include in your configuration?
  ⚡ invoice
  ⚡ notification
❯ ⚡ store
```

It automatically detects that we have used MongoDB and asks us the next question, that is, whether we would like to have MongoDB as a cluster. We won't choose anything here:

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Which *type* of gateway would you like to use? JHipster gateway based on Netflix Zuul
? Enter the root directory where your gateway(s) and microservices are located ../
3 applications found at /home/deepu/Documents/jhipster-book/v2/e-commerce-app/

? Which applications do you want to include in your configuration? invoice, notification, store
? Which applications do you want to use with clustered databases (only available with MongoDB and Couchbase)?
>X notification
```

Then, it asks us about monitoring; that is, whether we need to set up any monitoring for the application. We will choose logs and metrics with the JHipster Console (based on ELK and Zipkin):

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Which *type* of gateway would you like to use? JHipster gateway based on Netflix Zuul
? Enter the root directory where your gateway(s) and microservices are located ../
3 applications found at /home/deepu/Documents/jhipster-book/v2/e-commerce-app/

? Which applications do you want to include in your configuration? invoice, notification, store
? Which applications do you want to use with clustered databases (only available with MongoDB and Couchbase)?
? Do you want to setup monitoring for your applications ?
  No
> Yes, for logs and metrics with the JHipster Console (based on ELK and Zipkin)
  Yes, for metrics only with Prometheus
```

We can also opt out of the monitoring option or choose Prometheus. This connects with Prometheus and shows metrics only.

Then, JHipster asks us whether we need Curator or Zipkin:

- **Curator** will help you curate and manage the indices that are created by Elasticsearch.
- **Zipkin** provides distributed tracing, as we discussed in the previous chapter:

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Which *type* of gateway would you like to use? JHipster gateway based on Netflix Zuul
? Enter the root directory where your gateway(s) and microservices are located ../
3 applications found at /home/deepu/Documents/jhipster-book/v2/e-commerce-app/

? Which applications do you want to include in your configuration? invoice, notification, store
? Which applications do you want to use with clustered databases (only available with MongoDB and Couchbase)? (Press <space> to select,
? Do you want to setup monitoring for your applications ? Yes, for logs and metrics with the JHipster Console (based on ELK and Zipkin)
? You have selected the JHipster Console which is based on the ELK stack and additional technologies, which one do you want to use ?
  ○Curator, to help you curate and manage your Elasticsearch indices
> Zipkin, for distributed tracing (only compatible with JHipster >= v4.2.0)
```

We will only choose Zipkin here.



We can also choose nothing here and go with the default option.

Finally, it asks for the password for the JHipster Registry; we will go with the default option here:

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Which *type* of gateway would you like to use? JHipster gateway based on Netflix Zuul
? Enter the root directory where your gateway(s) and microservices are located ../
3 applications found at /home/deepu/Documents/jhipster-book/v2/e-commerce-app/

? Which applications do you want to include in your configuration? invoice, notification, store
? Which applications do you want to use with clustered databases (only available with MongoDB and Couchbase)? (Press <space> to select, <a> to toggle all, <i> to invert selection)
? Do you want to setup monitoring for your applications ? Yes, for logs and metrics with the JHipster Console (based on ELK and Zipkin)
? You have selected the JHipster Console which is based on the ELK stack and additional technologies, which one do you want to use ? Zipkin, for distributed tracing (only compatible with JHipster >= v4.2.0)
JHipster registry detected as the service discovery and configuration provider used by your apps
? Enter the admin password used to secure the JHipster Registry (admin) _
```

That's it; we have just created a higher-level Docker Compose setup that contains information about all the services that we need in order to run the application successfully:

```
⌚ Welcome to the JHipster Docker Compose Sub-Generator ⌚
Files will be generated in folder: /home/deepu/Documents/jhipster-book/v2/e-commerce-app/docker-compose

Checking Docker images in applications directories...
  create log-conf/logstash.conf
  create log-data/.gitignore
  create docker-compose.yml
  create README-DOCKER-COMPOSE.md
  create jhipster-registry.yml
  create central-server-config/application.yml
  create jhipster-console.yml

Docker Compose configuration successfully generated!
You can launch all your infrastructure by running : docker-compose up -d
INFO! Congratulations, JHipster execution is complete!
```

Now, we can run the entire suite with the following command:

```
> docker-compose up -d
```

This will start the store gateway, notification, invoice, and the registry, along with the JHipster Console and all the other required services.

Don't forget to commit the generated files to Git.

## Features of the deployed application

Now, the deployed applications are ready to be launched. We can launch the JHipster Registry at <http://localhost:8761>; it will list all the registered applications:

The screenshot shows the JHipster Registry interface. At the top, there's a navigation bar with links for Home, Eureka, Configuration, Administration, and a refresh button. Below the header, the title "JHipster Registry v5.0.2" is displayed. The main content area is divided into several sections:

- System Status:** Shows environment (JHipster-Environment), data center (JHipster-DataCenter), current time (2019-12-09T18:17:26 +0000), system uptime (00:17), and below renew threshold (false).
- Instances Registered:** A table showing registered applications (App) and their instance IDs. All three instances (INVOICE, NOTIFICATION, STORE) are marked as "UP".

App	Instance ID	Status
INVOICE	invoice:b96380f9b3ed6a5d12f003bbc989eaf7	UP
NOTIFICATION	notification:966ce6cb1d1e66bb825d3307b44e975a	UP
STORE	store:c2927a100d855cc00640007b33c7c2b7	UP
- General Info:** Displays total available memory (353mb), current memory usage (101mb (28%)), number of CPU (12), instance IP address (172.19.0.3), and instance status (UP).
- Health:** A table showing the status of various components: DiskSpace, RefreshScope, DiscoveryComposite, ConfigServer, and Hystrix, all marked as "UP".

Component	Status
DiskSpace	UP
RefreshScope	UP
DiscoveryComposite	UP
ConfigServer	UP
Hystrix	UP

At the bottom left, a footer note says "JHipster Registry - [www.jhipster.tech](http://www.jhipster.tech)".

Added to that, the registry also tells us about the number of instances that have been registered. Navigate to **Eureka | Instances** to check that. Currently, one of each instance has been registered:

The screenshot shows the "Application Instances" page. It lists three registered applications: INVOICE, NOTIFICATION, and STORE. Each application entry shows a count of 1/1 instances.

INVOICE	1/1
NOTIFICATION	1/1
STORE	1/1

Similarly, the gateway application will list the microservices that are connected to it. Go to <http://localhost:8080>.

Log in and navigate to **Administration | Gateway** to see the microservice applications that are connected to this gateway application:

The screenshot shows the JHipster Admin UI interface. At the top, there's a navigation bar with links for Home, Entities, Administration, Language, and Account. Below the navigation is a header for 'Store v0.0.1-SNAPSHOT'. The main content area is titled 'Gateway' and 'Current routes'. It lists two services: 'notification' and 'invoice'. For each service, it shows the URL pattern ('/services/notification/\*\*' and '/services/invoice/\*\*'), the service name ('notification' and 'invoice'), and the available servers. Each server entry includes an 'UP' status indicator, a color-coded健康状态 (green for 'prod', blue for 'dev', grey for 'staging', red for 'test'), the server name ('master' or 'primary'), and the port ('8082' or '8081'). Below the table, there's a footer message: 'This is your footer'.

Here, the gateway has registered our invoice and notification applications.

## JHipster Console demo

JHipster also provides a console application based on the Elastic Stack (often referred to as ELK), which can be used for logs and metrics monitoring of the application. JHipster Console is another open source application. It is really useful and provides some nice dashboards that can be used to visualize the application. As with other JHipster products, it is much easier to get started with the JHipster Console.



The JHipster Console project is available in GitHub (<https://github.com/jhipster/jhipster-console>).

Our applications stream the metrics and logs into the JHipster console. To make that happen, there are a few settings in the `application-prod.yml` file in all the applications (gateway and microservices applications):

```
metrics:  
  logs: # Reports metrics in the logs  
    enabled: false  
    report-frequency: 60 # in seconds  
logging:  
  use-json-format: false # By default, logs are not in Json format  
  logstash: # Forward logs to logstash over a socket, used by  
    LoggingConfiguration  
    enabled: false
```

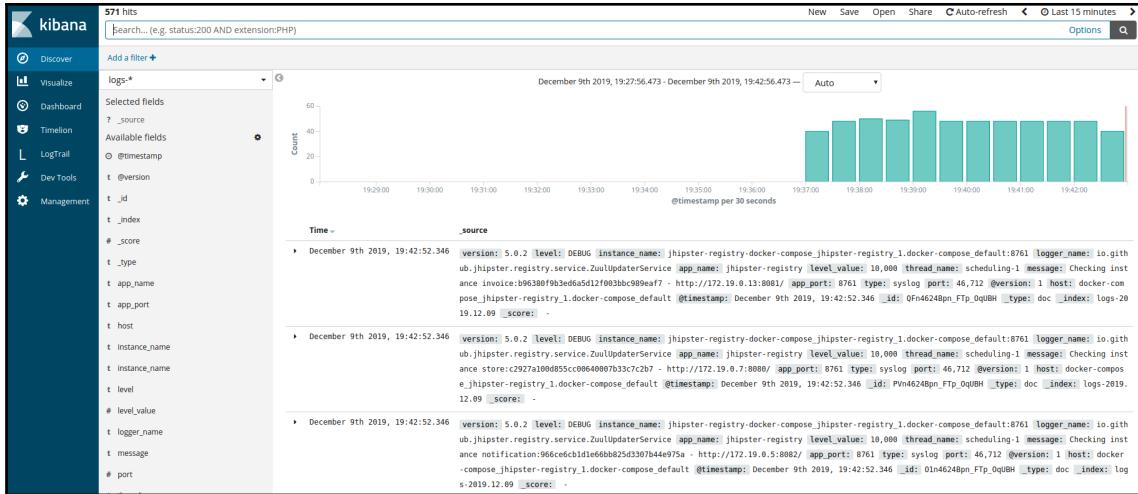
```
host: localhost
port: 5000
queue-size: 512
```

The `metrics.logs.enabled` and `logging.logstash.enabled` properties must be set to `true`. However, since we generated the `docker-compose` setup using JHipster, it created a file called `docker-compose/central-server-config/application.yml`, which will be used by the config server to override the preceding settings. This will push the logs to the console application. The JHipster Console will collect this information and show it in nice-looking dashboards with the help of Kibana. The following is the generated config file that's used by the config server:

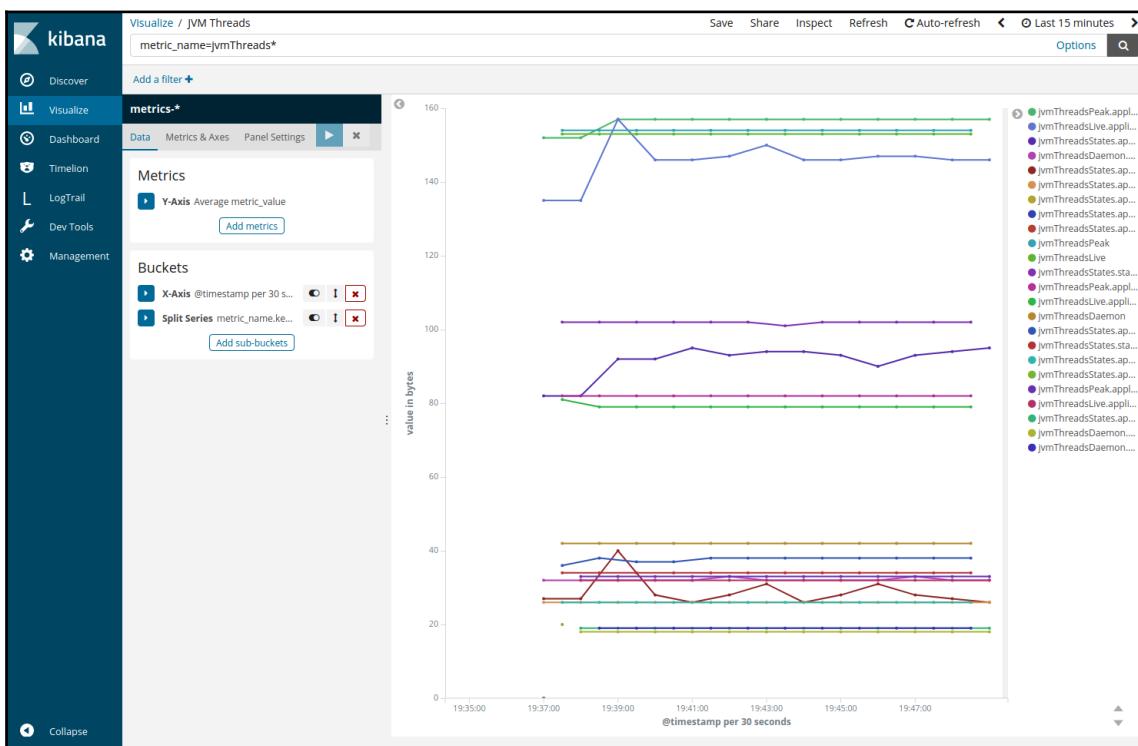
```
...
jhipster:
  security:
    authentication:
      jwt:
        base64-secret: ...
  logging:
    logstash: # forward logs to ELK
      enabled: true
      host: jhipster-logstash
  metrics:
    logs: # report metrics in the logs
      enabled: true
      report-frequency: 60 # in seconds
  spring:
    zipkin:
      base-url: http://jhipster-zipkin:9411
      enabled: true
    sleuth:
      sampler:
        probability: 1 # report 100% of traces to Zipkin
  eureka:
    client:
      service-url:
        defaultZone: http://admin:${{jhipster.registry.password}}@jhipster-registry:8761/eureka/
```

Let's look at the JHipster Console running on `http://localhost:5601`.

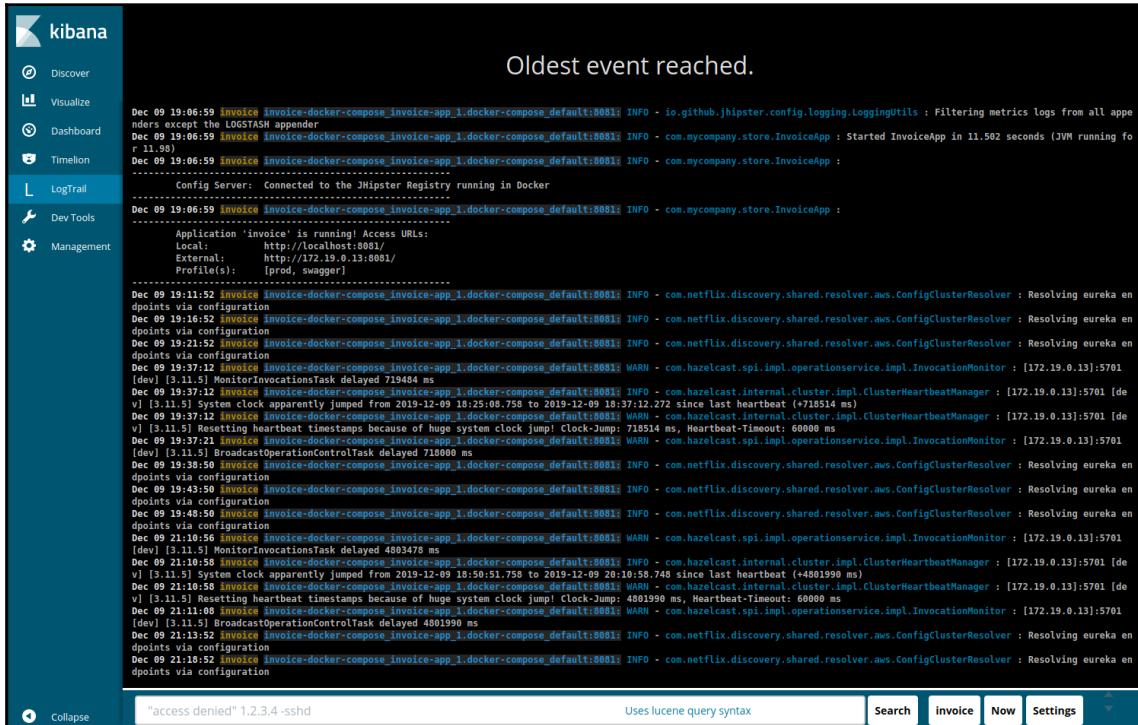
The JHipster Console provides the following (customizable) dashboards in Kibana:



It also provides preconfigured dashboards for logs, application-level metrics such as JVM threads metrics, and other details:



Added to this, the console also has an interface where we can see the application logs. It shows the log of all the applications we've deployed. We can filter and search the logs with respect to the application:



The screenshot shows the Kibana interface with the 'LogTrail' tab selected. The left sidebar includes links for Discover, Visualize, Dashboard, Timelion, Dev Tools, and Management. The main area displays log entries for the 'invoice' service. A message at the top says 'Oldest event reached.' Below are several log entries from December 9, 2019, at 19:06:59. The logs show various system and application events, such as connections to the JHipster Registry and Eureka endpoints, and a warning about a system clock jump.

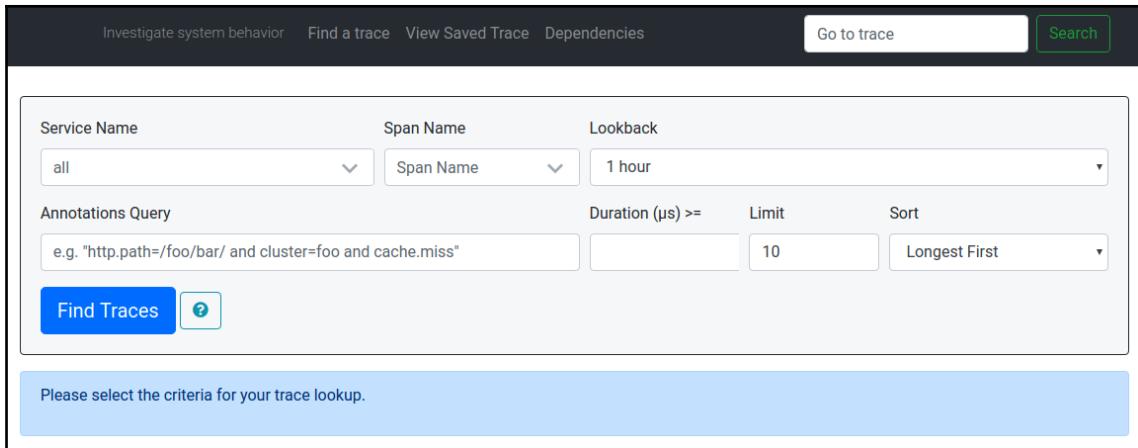
```

Dec 09 19:06:59 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - io.github.jhipster.config.logging.LoggingUtils : Filtering metrics logs from all appenders except the LOGSTASH appender
Dec 09 19:06:59 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.mycompany.store.InvoiceApp : Started InvoiceApp in 11.502 seconds (JVM running for 11.98)
Dec 09 19:06:59 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.mycompany.store.InvoiceApp :
-----
Config Server: Connected to the JHipster Registry running in Docker
-----
Dec 09 19:06:59 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.mycompany.store.InvoiceApp :
-----
Application 'invoice' is running! Access URLs:
Local: http://localhost:8081/
External: http://172.19.0.13:8081/
For file(s): [path, ...]
-----
Dec 09 19:11:52 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.netflix.discovery.shared.resolver.aws.ConfigClusterResolver : Resolving eureka endpoints via configuration
Dec 09 19:16:52 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.netflix.discovery.shared.resolver.aws.ConfigClusterResolver : Resolving eureka endpoints via configuration
Dec 09 19:21:52 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.netflix.discovery.shared.resolver.aws.ConfigClusterResolver : Resolving eureka endpoints via configuration
Dec 09 19:27:12 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: WARN - com.hazelcast.spi.impl.operationservice.impl.InvocationMonitor : [172.19.0.13]:5701 [dev] [3.11.5] MonitorInvolcationsTask delayed 719484 ms
Dec 09 19:37:12 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.hazelcast.internal.cluster.impl.ClusterHeartbeatManager : [172.19.0.13]:5701 [dev] [3.11.5] System clock apparently jumped from 2019-12-09 18:25:08.758 to 2019-12-09 18:37:12.272 since last heartbeat (+718514 ms)
Dec 09 19:37:12 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: WARN - com.hazelcast.internal.cluster.impl.ClusterHeartbeatManager : [172.19.0.13]:5701 [dev] [3.11.5] Resetting heartbeat timestamps because of huge system clock jump! Clock-Jump: 789514 ms, Heartbeat-Timeout: 60000 ms
Dec 09 19:37:12 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.hazelcast.spi.impl.operationservice.impl.InvocationMonitor : [172.19.0.13]:5701 [dev] [3.11.5] BroadcastOperationControlTask delayed 718000 ms
Dec 09 19:38:50 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.netflix.discovery.shared.resolver.aws.ConfigClusterResolver : Resolving eureka endpoints via configuration
Dec 09 19:43:50 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.netflix.discovery.shared.resolver.aws.ConfigClusterResolver : Resolving eureka endpoints via configuration
Dec 09 19:48:50 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.netflix.discovery.shared.resolver.aws.ConfigClusterResolver : Resolving eureka endpoints via configuration
Dec 09 19:58:56 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: WARN - com.hazelcast.spi.impl.operationservice.impl.InvocationMonitor : [172.19.0.13]:5701 [dev] [3.11.5] MonitorInvolcationsTask delayed 4803478 ms
Dec 09 21:10:58 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.hazelcast.internal.cluster.impl.ClusterHeartbeatManager : [172.19.0.13]:5701 [dev] [3.11.5] System clock apparently jumped from 2019-12-09 18:50:51.758 to 2019-12-09 20:10:58.748 since last heartbeat (+4801990 ms)
Dec 09 21:10:58 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: WARN - com.hazelcast.internal.cluster.impl.ClusterHeartbeatManager : [172.19.0.13]:5701 [dev] [3.11.5] Resetting heartbeat timestamps because of huge system clock jump! Clock-Jump: 4801990 ms, Heartbeat-Timeout: 60000 ms
Dec 09 21:13:52 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.hazelcast.spi.impl.operationservice.impl.InvocationMonitor : [172.19.0.13]:5701 [dev] [3.11.5] BroadcastOperationControlTask delayed 4801990 ms
Dec 09 21:13:52 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.netflix.discovery.shared.resolver.aws.ConfigClusterResolver : Resolving eureka endpoints via configuration
Dec 09 21:18:52 invoice invoice-docker-compose_invoice-app_1.docker-compose_default:8081: INFO - com.netflix.discovery.shared.resolver.aws.ConfigClusterResolver : Resolving eureka endpoints via configuration

```

At the bottom of the interface, there are buttons for 'Collapse', a search bar containing 'access denied' 1.2.3.4 -sshd', and filters for 'Search', 'invoice', 'Now', and 'Settings'.

Since we chose Zipkin integration, the Zipkin server is also configured and the trace details can be viewed in its UI, which is provided as part of the JHipster Console and integrates with the Kibana dashboard, at <http://127.0.0.1:9411/>, or from the trace data that was integrated into the Kibana dashboard in the JHipster Console:



Zipkin tracing is disabled by default. To enable Zipkin tracing, we need to include the `zipkin` profile when we build our images and need to set the `spring.zipkin.enabled` property to `true` in the application configurations. The configuration is enabled when running the preceding Docker Compose setup, but the images still need to be built with the `zipkin` Gradle profile. This will trigger span reporting to the Zipkin server and also add correlation IDs (`traceID`, `spanID`, and `parentID`) to request headers and logs.

You can learn more about the JHipster Console by visiting <https://www.jhipster.tech/monitoring/#jhipster-console>.

## Scaling up with Docker

Docker Compose gives us the flexibility to scale our application with a single command with `docker-compose`:

```
> docker-compose up --scale <app-name>=<number of instances>
```

We can scale the instances using the following command:

```
> docker-compose up --scale invoice-app=2
```

The preceding command will spin up another invoice instance. We can view this on the JHipster Registry dashboard, as follows:

The screenshot shows the JHipster Registry interface. At the top, there's a navigation bar with links for Home, Eureka, Configuration, Administration, and a refresh button. Below the navigation is a section titled "Application Instances" which lists three services: INVOICE (2/2), NOTIFICATION (1/1), and STORE (1/1). To the right of this section are two buttons: "Refresh now" and "disabled". Below this is a table titled "Instances" with columns for "ID" and "Status". It contains two rows of data, each representing an instance of the INVOICE service. The first row has the ID "invoice:b96380f9b3ed6a5d12f003bbc989eaf7" and the second row has the ID "invoice:763a7c49a3c4002e35ba8f1efcf1c251". Both rows show an "UP" status, "zone primary", "profile prod.swagger", "version 0.0.1-SNAPSHOT", "management port 8081", "git-commit 1e4c5ae", and "git-branch master". At the bottom left of the dashboard, there's a footer link: "JHipster Registry - [www.jhipster.tech](#)".

And that's it – we have successfully deployed and scaled our microservice application using Docker Compose.

## Summary

In this chapter, we learned how to generate, set up, and start the Docker Compose configurations for our microservices, JHipster Registry, and Console. We also looked at their features. Then, we briefly learned about Kubernetes, OpenShift, and Docker Compose. We also learned how to create Docker Compose configurations using JHipster. This was followed by how to scale the application with Docker Compose.

In the next chapter, we will learn how to deploy the application to Google Cloud using Kubernetes.

# 12

## Deploying to the Cloud with Kubernetes

Kubernetes is the de facto container orchestration tool in the cloud world. As we have seen in the previous chapter, it comes with many additional features and is easy to configure and manage. This makes Kubernetes a default choice for container orchestration. The ability to mask the lower-level details and provide out-of-the-box service discovery, self-healing, and health checks, among other features, attracted many companies and organizations to switch to Kubernetes.



Did you know that Kubernetes is the evolution of Google's internal orchestration tool called Borg?

In this chapter, we will cover the following topics:

- Generating Kubernetes configuration files with JHipster
- Walking through the generated files to learn about the Kubernetes configurations used
- Deploying the application to Google Cloud with Kubernetes
- Using Istio service mesh to learn how to use a service mesh with our application

# Generating Kubernetes configuration files with JHipster

Knowing the components of Kubernetes and how they work is beyond the scope of this book. However, we will look at how JHipster simplifies microservices deployment with Kubernetes. Let's go ahead and generate the Kubernetes configuration files.

To deploy applications with Kubernetes, you need the Kubernetes `kubectl` client. To install `kubectl`, please follow the instructions on the Kubernetes website: <https://kubernetes.io/docs/tasks/tools/install-kubectl/>.



The Kubernetes sub-generator needs `kubectl` (v1.2 or later) to be installed on your computer. `kubectl` is the command-line interface for Kubernetes.

We can also install `kubectl` along with the Cloud SDK from Google Cloud. Set up the `gcloud` CLI, as follows:

1. Download the binary (based on your operating system) from <https://cloud.google.com/sdk/install>.
2. Install the application by following the steps given on the website (make sure that you have Python installed).
3. Once installed, set up Google Cloud. In order to set up Google Cloud, run `gcloud init`.
4. This will then ask you to log in to your Google account; do so to authenticate (you must have a valid Google Cloud account).
5. Run `gcloud components install kubectl` to install the Kubernetes client.

## Generating the Kubernetes manifests

Similar to the Docker Compose sub-generator, JHipster also comes bundled with a Kubernetes sub-generator. In order to use it, just like with Docker Compose, we will create a new folder and name it Kubernetes. Then, we will go inside the folder to create the configuration files.

We can create Kubernetes configuration files with the following command:

```
> mkdir kubernetes && cd kubernetes  
> jhipster kubernetes
```



As we have seen already, Kubernetes needs separate tools for running locally (that is, for development purposes). Therefore, if you need to do things locally, please enable the local Kubernetes cluster using Docker Desktop for Mac (<https://docs.docker.com/docker-for-mac/kubernetes/>) or Windows (<https://docs.docker.com/docker-for-windows/kubernetes/>). For Linux, install Minikube (<https://kubernetes.io/docs/setup/learning-environment/minikube/>) from Kubernetes.

We then need to answer the questions that the sub-generator asks us, as follows:

- The first question the sub-generator asks is what type of application we'd like to deploy. It provides two options: monolithic and microservices. We will choose the microservices option:

```
* Welcome to the JHipster Kubernetes Generator *
Files will be generated in folder: /home/deepu/Documents/jhipster-book/v2/e-commerce-app/kubernetes
✓ Docker is installed
? Which *type* of application would you like to deploy? (Use arrow keys)
❯ Monolithic application
  Microservice application
```

- Then, it asks us to enter the root directory. We will select the default option since our directories are present as the siblings of the Kubernetes folder:

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Enter the root directory where your gateway(s) and microservices are located (...) _
```

- Then, the sub-generator will list all the folders with our JHipster-generated applications. Here, it will list all three applications that we need—store, invoice, and notification. Select all three applications and hit *Enter*:

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Enter the root directory where your gateway(s) and microservices are located ../
3 applications found at /home/deepu/Documents/jhipster-book/v2/e-commerce-app/

? Which applications do you want to include in your configuration?
  ◉ invoice
  ◉ notification
❯◉ store
```

- Then, it will ask whether we need monitoring. The options are JHipster Console or Prometheus, and we will select JHipster Console:

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Enter the root directory where your gateway(s) and microservices are located ../
3 applications found at /home/deepu/Documents/jhipster-book/v2/e-commerce-app/
? Which applications do you want to include in your configuration?
  • invoice
  • notification
  >• store
```

- Next, it will ask whether we need clustering enabled for MongoDB, similar to the docker-compose sub-generator. Let's skip clustering and proceed. Just hit *Enter*:

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Enter the root directory where your gateway(s) and microservices are located ../
3 applications found at /home/deepu/Documents/jhipster-book/v2/e-commerce-app/
? Which applications do you want to include in your configuration? invoice, notification, store
? Do you want to setup monitoring for your applications ? Yes, for logs and metrics with the JHipster Console
? Which applications do you want to use with clustered databases (only available with MongoDB and Couchbase)?
Xnotification
```

- Then, it will ask us to provide the password for the registry service. In our case, it is JHipster Registry. We will select the default one for now, but it is generally advisable to use a strong password here:

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Enter the root directory where your gateway(s) and microservices are located ../
3 applications found at /home/deepu/Documents/jhipster-book/v2/e-commerce-app/
? Which applications do you want to include in your configuration? invoice, notification, store
? Do you want to setup monitoring for your applications ? Yes, for logs and metrics with the JHipster Console
? Which applications do you want to use with clustered databases (only available with MongoDB and Couchbase)?
JHipster registry detected as the service discovery and configuration provider used by your apps
? Enter the admin password used to secure the JHipster Registry (admin) _
```

- Afterward, it will ask us for the namespace that we need to use in Kubernetes.

So, what is a namespace? We can consider namespaces as a group within which resources should be named uniquely. When the cluster is shared between different users or teams, namespaces can provide resource quotas for them. Ideally, namespaces should be used only for a larger team. For smaller teams, it is better to go with default options. Kubernetes, by default, provides three namespaces, which are as follows:

- **default**: When you start a container or pod without providing any namespaces, they will end up in the default namespace.
- **kube-system**: This namespace contains Kubernetes system-based objects.
- **kube-admin**: This is a public namespace, which will be shown to all the users publically without any authentication.

We will set `jhipster` as the namespace here for the sake of this demo. Let's see what else the sub-generator asks us:

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Enter the root directory where your gateway(s) and microservices are located ../
3 applications found at /home/deepu/Documents/jhipster-book/v2/e-commerce-app/
? Which applications do you want to include in your configuration? invoice, notification, store
? Do you want to setup monitoring for your applications ? Yes, for logs and metrics with the JHipster Console
? Which applications do you want to use with clustered databases (only available with MongoDB and Couchbase)?
JHipster registry detected as the service discovery and configuration provider used by your apps
? Enter the admin password used to secure the JHipster Registry admin
? What should we use for the Kubernetes namespace? jhipster_
```

- The sub-generator will ask for our Docker repository name so that Kubernetes can use this Docker repository to pull the images (the login username of the Docker repository, or the URL in case of a custom Docker Registry):

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Enter the root directory where your gateway(s) and microservices are located ../
3 applications found at /home/deepu/Documents/jhipster-book/v2/e-commerce-app/
? Which applications do you want to include in your configuration? invoice, notification, store
? Do you want to setup monitoring for your applications ? Yes, for logs and metrics with the JHipster Console
? Which applications do you want to use with clustered databases (only available with MongoDB and Couchbase)?
JHipster registry detected as the service discovery and configuration provider used by your apps
? Enter the admin password used to secure the JHipster Registry admin
? What should we use for the Kubernetes namespace? jhipster_
? What should we use for the base Docker repository name? deepu105_
```

- Then, the sub-generator will ask for the command that needs to be used to push the image to the Docker repository. We will select the default command here:

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Enter the root directory where your gateway(s) and microservices are located ../
3 applications found at /home/deepu/Documents/jhipster-book/v2/e-commerce-app/

? Which applications do you want to include in your configuration? invoice, notification, store
? Do you want to setup monitoring for your applications ? Yes, for logs and metrics with the JHipster Console
? Which applications do you want to use with clustered databases (only available with MongoDB and Couchbase)?
JHipster registry detected as the service discovery and configuration provider used by your apps
? Enter the admin password used to secure the JHipster Registry admin
? What should we use for the Kubernetes namespace? jhipster
? What should we use for the base Docker repository name? deepu105
? What command should we use for push Docker image to repository? (docker push) _
```

- Then, it will ask whether we want to use Istio. We will see what Istio is and how to use it later in this chapter. For now, choose No:

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Enter the root directory where your gateway(s) and microservices are located ../
3 applications found at /home/deepu/Documents/jhipster-book/v2/e-commerce-app/

? Which applications do you want to include in your configuration? invoice, notification, store
? Do you want to setup monitoring for your applications ? Yes, for logs and metrics with the JHipster Console
? Which applications do you want to use with clustered databases (only available with MongoDB and Couchbase)?
JHipster registry detected as the service discovery and configuration provider used by your apps
? Enter the admin password used to secure the JHipster Registry admin
? What should we use for the Kubernetes namespace? jhipster
? What should we use for the base Docker repository name? deepu105
? What command should we use for push Docker image to repository? docker push
? Do you want to enable Istio? (Use arrow keys)
❯ No
Yes
```

- Then, the generator will ask us to choose the Kubernetes service type. So, what is the service type?

In Kubernetes, everything that we deploy is a pod. These pods are managed by replication controllers that can create and destroy any pods. Each pod needs an identifier, so they are tagged with an IP address. This dynamic nature of pods will lead to a lot of problems for other pods that depend on them. As a solution to this problem, Kubernetes introduced services. Services are nothing but a logical grouping of unique pods that have policies attached to them. These policies are applicable for all the pods inside the services, but we need to publish these services to the external world to access them.



One of the most powerful features of Kubernetes is that it helps to maintain the number of pod replicas consistently. The replication controller helps to maintain the pod count by automatically shutting down and booting up the pods.

Kubernetes gives us four different service types, as follows:

- **ClusterIP:** This is the default type. This will assign the cluster's internal IP and make it visible within the cluster itself.
- **NodePort:** This will expose the service to a static port in the node's IP. The port will be random and will be chosen between 30000-32767.
- **LoadBalancer:** This will expose the service externally. Kubernetes will assign an IP automatically. This will create a route to the `NodePort` and cluster IP internally.
- **Ingress:** This is a special option that Kubernetes provides. This will provide load balancing, SSL termination, and name-based virtual hosting to the services.

We will select the `LoadBalancer` option:

```
✓ Docker is installed
? Which *type* of application would you like to deploy? Microservice application
? Enter the root directory where your gateway(s) and microservices are located ../
3 applications found at /home/deepu/Documents/jhipster-book/v2/e-commerce-app/

? Which applications do you want to include in your configuration? invoice, notification, store
? Do you want to setup monitoring for your applications ? Yes, for logs and metrics with the JHipster Console
? Which applications do you want to use with clustered databases (only available with MongoDB and Couchbase)?
JHipster registry detected as the service discovery and configuration provider used by your apps
? Enter the admin password used to secure the JHipster Registry admin
? What should we use for the Kubernetes namespace? jhipster
? What should we use for the base Docker repository name? deepu105
? What command should we use for push Docker image to repository? docker push
? Do you want to enable Istio? No
? Choose the Kubernetes service type for your edge services (Use arrow keys)
❯ LoadBalancer - Let a Kubernetes cloud provider automatically assign an IP
  NodePort - expose the services to a random port (30000 - 32767) on all cluster nodes
  Ingress - create ingresses for your services. Requires a running ingress controller
```

That's it. This will generate the necessary configuration files for us to deploy the application with Kubernetes and will print out the commands needed for the next steps:

```
Kubernetes configuration successfully generated!

WARNING! You will need to push your image to a registry. If you have not done so, use the following commands to tag and push the images:
docker image tag invoice deepu105/invoice
docker push deepu105/invoice
docker image tag notification deepu105/notification
docker push deepu105/notification
docker image tag store deepu105/store
docker push deepu105/store

INFO! Alternatively, you can use Jib to build and push image directly to a remote registry:
./gradlew bootJar -Pprod jibBuild -Djib.to.image=deepu105/invoice in /home/deepu/Documents/jhipster-book/v2/e-commerce-app/invoice
./gradlew bootJar -Pprod jibBuild -Djib.to.image=deepu105/notification in /home/deepu/Documents/jhipster-book/v2/e-commerce-app/notification
./gradlew bootJar -Pprod jibBuild -Djib.to.image=deepu105/store in /home/deepu/Documents/jhipster-book/v2/e-commerce-app/store

You can deploy all your apps by running the following script:
bash kubectl-apply.sh

Use these commands to find your application's IP addresses:
kubectl get svc store -n jhipster

INFO! Congratulations, JHipster execution is complete!
```

Next up, we will check the files that have been generated.

## Walking through the generated files

The files generated by JHipster are organized by application. That is, each application will have its own folder and the files related to that service will be present inside it.

We will start with the store gateway application. There will be three generated files: the `store-service.yml`, `store-mysql.yml`, and `store-deployment.yml` files.

The following is the `store-service.yml` file:

```
apiVersion: v1
kind: Service
metadata:
  name: store
  namespace: jhipster
  labels:
    app: store
spec:
  selector:
    app: store
    type: LoadBalancer
  ports:
    - name: http
      port: 8080
```

The first line defines the API version of Kubernetes we want to target, followed by the kind of template or object that this template carries. This template has a service defined in it.

Then, we have the metadata information. Kubernetes uses this metadata information to group certain services together. In the metadata, we can define the following:

- The service name
- The namespace the object belongs to
- The labels, which are key and value pairs

Then, we have the spec. The spec in the Kubernetes object will provide the state of the service. In the spec, we can define the number of replicas we need. We also have the selector, within which we specify the deployment with identifiers (we will see the deployment spec soon). We also specify the type of service, followed by the ports in which the application should run. This is similar to the Dockerfile, so we are exposing the 8080 port for the gateway service.

Then, we have the `store-mysql.yml` file, where we have defined our MySQL server for the store application. The difference here is that the service spec points to `store-mysql`, which is defined in the same file and is exposed on port 3306:

```
apiVersion: v1
kind: Service
metadata:
  name: store-mysql
  namespace: jhipster
spec:
  selector:
    app: store-mysql
  ports:
    - port: 3306
```

In the `store-mysql` app declaration, as shown in the next snippet, we have specified the database and environment properties that are needed for our application to run. Here, the kind is mentioned as **Deployment**. The job of the deployment object is to change the state of the services to the state that is defined in the deployment object.

Here, we have defined a single replica of the MySQL server, followed by the spec where we have mentioned the version of MySQL that we need (the container).



When it comes to databases, it is often preferable to use external database services rather than having the database in Kubernetes to reduce complexity.

This is then followed by the environment where we have the username, password, and then the database schema. We also have the volume information with volume mounts for persistent storage.

We can also define a spec inside a spec object (as shown in the following code):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
spec:
  replicas: 1
  selector:
    matchLabels:
      app: store-mysql
  template:
    metadata:
      labels:
        app: store-mysql
    spec:
      ...
      containers:
        - name: mysql
          image: mysql:8.0.18
          env:
            ...
          args:
            ...
          ports:
            - containerPort: 3306
      volumeMounts:
        - name: data
          mountPath: /var/lib/mysql/
```

Similarly, we have `store-deployment.yaml`, in which we have defined the store gateway application and its environment properties, along with the other details such as initialization containers, ports, resource limits, probes, and so on:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
spec:
  replicas: 1
  selector:
    ...
  template:
```

```
...
spec:
  initContainers:
    - name: init-ds
      image: busybox:latest
      command: # This waits for DB to be ready
      ...
  containers:
    - name: store-app
      image: deepu105/store
      env:
        - name: SPRING_PROFILES_ACTIVE
          value: prod
        ...
        - name: JHIPSTER_REGISTRY_PASSWORD
          valueFrom:
            secretKeyRef:
              name: registry-secret
              key: registry-admin-password
      ...
  resources:
    requests:
      ...
    limits:
      ...
  ports:
      ...
  readinessProbe:
      ...
  livenessProbe:
      ...
  ...
```

A similar approach is used for both the invoice and notification services. You can find them in their respective folders.

In JHipster-registry, alongside Service and Deployment, we have defined a Secret and a StatefulSet.

The secret is used to handle passwords. It will be an opaque type and the password is Base64-encoded.

Then, we have `StatefulSet`, which is similar to a pod except it has a sticky identity. Pods are dynamic in nature; these pods have a persistent identifier that is maintained throughout. It makes sense for a registry server to be defined as `StatefulSet` since it is essential that the registry server should be identified by a persistent identifier. This enables all services to connect to that single endpoint and get the necessary information. If the registry server is down, then communication between the services will also have problems since the services connect to other services via the registry server.

There are various options that can be set for the controller, which are as follows:

- **Replica set:** This provides a replica of pods at any time with selectors.
- **Replica controller:** This provides a replica of pods without any selectors.
- **StatefulSet:** This makes the pod unique by providing it with a persistent identifier.
- **DaemonSet:** This provides a copy of the pod that is going to be run.

The JHipster Registry is configured in a cluster with high availability. The UI access to the JHipster Registry is also restricted to the cluster for better security.

Similarly, configuration files are generated for the JHipster Console, and they are placed in a `jhipster-console.yml` folder where the JHipster Console is also defined.

The JHipster Console runs on an **Elastic (ELK) Stack**, so we need Elasticsearch, which is defined in `jhipster-elasticsearch.yml`, followed by Logstash in the `jhipster-logstash.yml` file.

Commit the generated files to Git.

Now, let's see how we can deploy this.

## Deploying the application to Google Cloud with Kubernetes

We have created Kubernetes configuration files with the `jhipster kubernetes` command. The next step is to build the artifacts and deploy them into Google Cloud.



It is also possible to deploy to other Kubernetes services such as Azure Kubernetes Service or Amazon Elastic Kubernetes Service using this configuration. Just follow the cloud provider's documentation to create a Kubernetes cluster and apply the generated configuration using the `kubectl apply` commands, as mentioned later in this section.

Kubernetes will use the image from the Docker Registry. We configured the Docker username when we generated the application, so the first step will be to tag those images and then push them to our Docker repository.

To do so, we will do the following:

1. Open the Terminal and go to the Kubernetes folder that we have generated. We will tag the images first:

```
> docker image tag store deepu105/store
```

2. Next, we will push this image into the Docker repository:

```
> docker push deepu105/store
```

Alternatively, if we haven't built the Docker images already, we could do the build and push in a single step using the following command, which can be executed under each application folder (note the app name at the end):

```
> ./gradlew bootJar -Pprod jib -Djib.to.image=deepu105/store
```



Note: You have to log in to Docker Hub before pushing the image. You can log in to Docker using the `docker login` command, followed by your username and password. If you don't have an account, you can create one at the following link: <https://cloud.docker.com/>.

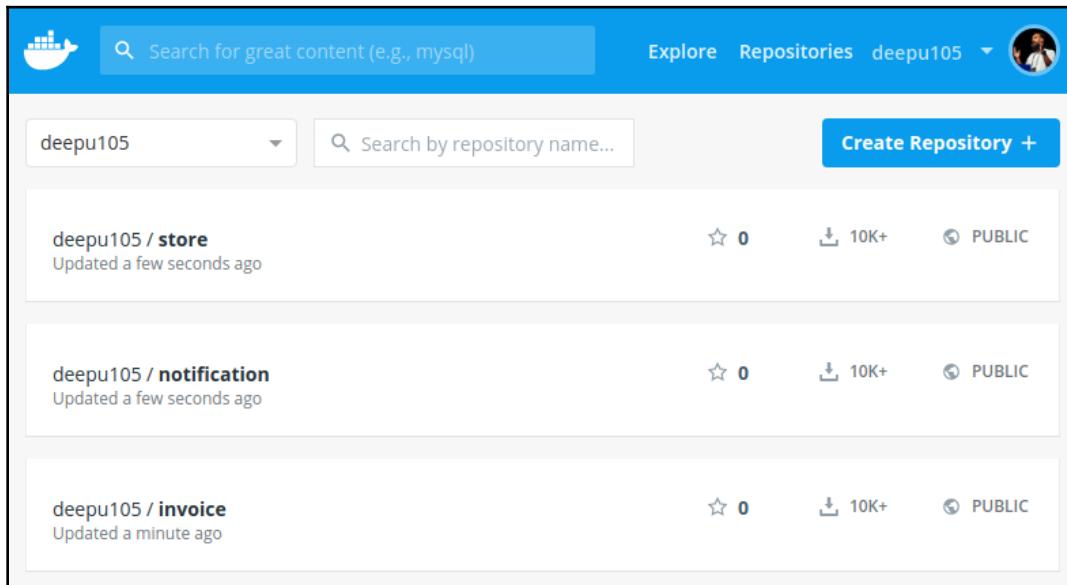
3. Similarly, push the `invoice` application to the Docker repository:

```
> docker image tag invoice deepu105/invoice  
> docker push deepu105/invoice
```

4. Do the same for notification:

```
> docker image tag notification deepu105/notification  
> docker push deepu105/notification
```

This will push the **store**, **invoice**, and **notification** to the Docker repository. We can check this in the Docker Console:



5. Now, we can connect to `gcloud` and deploy our containers with Kubernetes.



This assumes that you have set up the `gcloud` SDK and `kubectl` on your machine.

6. First, we will log in to the `gcloud` CLI via the Terminal. In order to do that, open your Terminal:

```
> gcloud init // if this is the first time you are using gcloud  
(Ignore this step if you logged in already)
```

Then, `gcloud` will ask you to log in to your Google account. Once validated, this will list the projects that you might already have.

Here, we can choose **[2] Create a new project** by entering the number before creating a new project. Then, press *Enter*. It will ask you to enter the project information and then configure a few Google services for that project. Then, `gcloud` will list all the available regions and you can choose a region that suits you.

7. If you have already logged in to the console and used it for other projects, then you can switch projects using the following command:

```
> gcloud config set project <project-name>
```

This will set the project, region, and the setting chosen as the default.

8. Then, you have to enable Kubernetes in your application. We can do this by logging in to our Google Cloud Console via the browser. Then, select the project that we have just created and go to <https://console.cloud.google.com/kubernetes/list> to create a new cluster. Please choose the CPU type **n1-standard-2** as we need a bit more CPU to deploy everything.
9. This will create a cluster for your project. Alternatively, you can also create a cluster using the `gcloud` command:

```
> gcloud container clusters create online-store-app \
--machine-type n1-standard-2
```

The following is the output of the preceding command:

```
Created [https://container.googleapis.com/v1/projects/jhipster-demo-deepu/zones/europe-west1-b/clusters/online-store-app].
To inspect the contents of your cluster, go to: https://console.cloud.google.com/kubernetes/workload_/gcloud/europe-west1-b/online-store-app?project=jhipster-demo-deepu
kubecfg entry generated for online-store-app.
NAME          LOCATION      MASTER_VERSION  MASTER_IP        MACHINE_TYPE   NODE_VERSION    NUM_NODES  STATUS
online-store-app  europe-west1-b  1.13.11-gke.14  35.205.201.187  n1-standard-2  1.13.11-gke.14  3          RUNNING
```

Thus, the cluster is created with three nodes and the configuration is added to our `kubectl` config.

10. Then, we can go to our Kubernetes folder and start deploying the services using `kubectl`:

```
> kubectl apply -f namespace.yml
> kubectl apply -f registry/
> kubectl apply -f invoice/
> kubectl apply -f notification/
> kubectl apply -f store/
> kubectl apply -f console/
```

Alternatively, you can also run the bash script, `./kubectl-apply.sh`, which does what we mentioned previously

The output will be as follows:

```
❯ ./kubectl-apply.sh
namespace/jhipster created
configmap/application-config created
secret/registry-secret created
service/jhipster-registry created
statefulset.apps/jhipster-registry created
deployment.apps/invoice created
deployment.apps/invoice-mysql created
service/invoice-mysql created
service/invoice created
deployment.apps/notification created
configmap/notification-mongodb-config created
configmap/notification-mongodb-init created
statefulset.apps/notification-mongodb created
service/notification-mongodb created
service/notification created
deployment.apps/store created
deployment.apps/store-mysql created
service/store-mysql created
service/store created
deployment.apps/jhipster-console created
service/jhipster-console created
job.batch/jhipster-import-dashboards created
configmap/es-config created
statefulset.apps/jhipster-elasticsearch-master created
statefulset.apps/jhipster-elasticsearch-data created
deployment.apps/jhipster-elasticsearch-client created
service/jhipster-elasticsearch-discovery created
service/jhipster-elasticsearch-data created
service/jhipster-elasticsearch created
deployment.apps/jhipster-logstash created
service/jhipster-logstash created
deployment.apps/jhipster-zipkin created
service/jhipster-zipkin created

#####
Please find the below useful endpoints,
JHipster Console - http://jhipster-console.jhipster.
#####
```

This will deploy all the applications to the Google Cloud environment, under your project.

11. You can check the pod's deployment process using the following command on bash:

```
> watch kubectl get pods -n jhipster
```

This will list the status of the pods that are spinning up in a watch loop so it is updated at specified intervals:

Every 2.0s: kubectl get pods -n jhipster					
NAME	READY	STATUS	RESTARTS	AGE	
invoice-5df7fd6f86-9nqpm	0/1	Init:0/1	0	21s	
invoice-mysql-7b48d66f8f-dwj5k	1/1	Running	0	20s	
jhipster-console-55c8d8f8d8-jpwqs	0/1	ContainerCreating	0	17s	
jhipster-elasticsearch-client-7656547b47-skkrc	0/1	Init:0/1	0	16s	
jhipster-elasticsearch-data-0	0/1	PodInitializing	0	16s	
jhipster-elasticsearch-master-0	0/1	Init:1/2	0	17s	
jhipster-import-dashboards-mk6gb	0/1	Init:0/1	0	17s	
jhipster-logstash-7fdc74bd58-dkbxj	0/1	Init:0/1	0	15s	
jhipster-registry-0	1/1	Running	0	21s	
jhipster-registry-1	0/1	ContainerCreating	0	12s	
jhipster-zipkin-7477f5f499-pqlsq	0/1	ContainerCreating	0	15s	
notification-67fdb9f787-v9g25	0/1	Init:0/1	0	19s	
notification-mongodb-0	0/1	Init:0/3	0	19s	
store-5df8f645b7-nvxsr	0/1	Init:0/1	0	18s	
store-mysql-76b66c895c-t74m9	1/1	Running	0	18s	

12. Once the pods are in **Running** status, you can also get the logs of the pod using the following command:

```
> kubectl logs <name of the pod as shown above> -n jhipster
```

The following is the output:

2019-12-10 10:50:50.875 INFO 1 --- [           main] com.mycompany.store.StoreApp : Started StoreApp in 115.946 seconds (JVM running for 118.534)
2019-12-10 10:50:50.891 INFO 1 --- [           main] com.mycompany.store.StoreApp :
Application 'store' is running! Access URLs:
Local:          http://localhost:8080/
External:       http://10.8.1.6:8080/
Profile(s):    [prod]
2019-12-10 10:50:50.892 INFO 1 --- [           main] com.mycompany.store.StoreApp :
-----
Config Server: Connected to the JHipster Registry running in Kubernetes
-----
2019-12-10 10:50:56.457 INFO 1 --- [trap-executor-0] c.n.d.s.r.aws.ConfigClusterResolver : Resolving eureka endpoints via configuration
2019-12-10 10:51:11.458 INFO 1 --- [trap-executor-0] c.n.d.s.r.aws.ConfigClusterResolver : Resolving eureka endpoints via configuration

You can stream the log by appending the `-f` flag to the command.

13. You can get the application's external IP using this command:

```
> kubectl get svc store -n jhipster
```

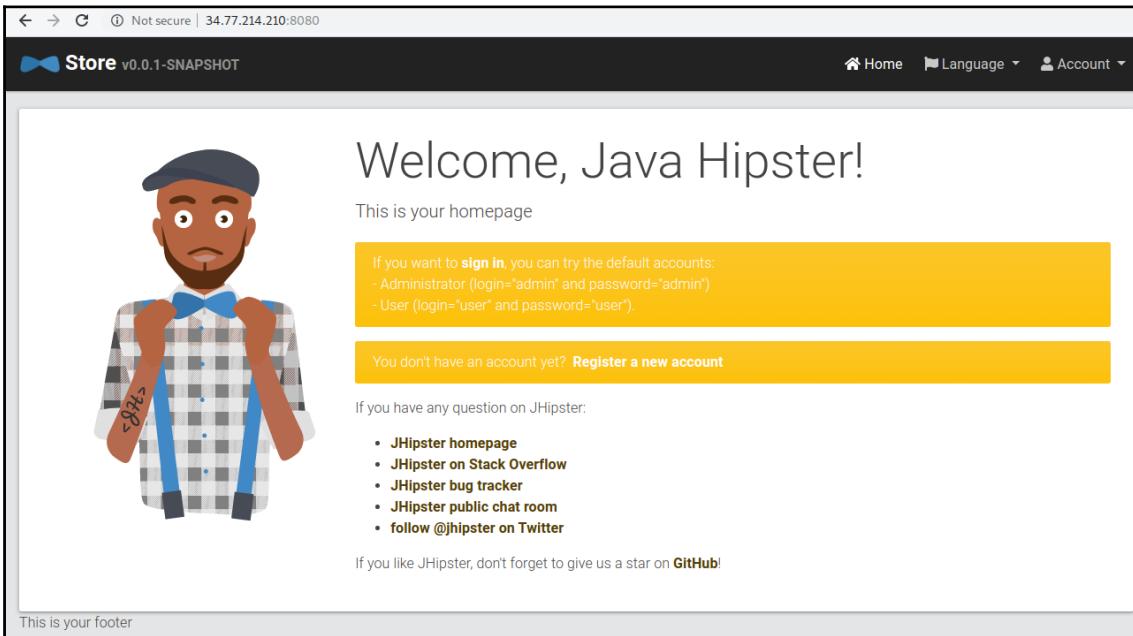
This will list the application name, type, IP address, external address, ports, and uptime:

```
~/Do...ts/jh...ok/v2/e...pp/kubernetes on ✘ master? □ 2
> kubectl get svc store -n jhipster
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
store     LoadBalancer   10.11.248.230   34.77.214.210   8080:32115/TCP   6m58s
```

We can find the same information on the Google Cloud Console as well under the **Services & Ingress** section:

Name	Status	Type	Endpoints	Pods	Namespace	Cluster
invoice	OK	Cluster IP	10.11.240.101	1 / 1	jhipster	online-store-app
invoice-mysql	OK	Cluster IP	10.11.244.116	1 / 1	jhipster	online-store-app
jhipster-console	OK	Load balancer	34.77.240.119:5601	1 / 1	jhipster	online-store-app
jhipster-elasticsearch	OK	Cluster IP	10.11.249.107	1 / 1	jhipster	online-store-app
jhipster-elasticsearch-data	OK	Cluster IP	10.11.242.10	1 / 1	jhipster	online-store-app
jhipster-elasticsearch-discovery	OK	Cluster IP	10.11.241.170	1 / 1	jhipster	online-store-app
jhipster-logstash	OK	Cluster IP	10.11.244.40	1 / 1	jhipster	online-store-app
jhipster-registry	OK	Cluster IP	None	2 / 2	jhipster	online-store-app
jhipster-zipkin	OK	Cluster IP	10.11.252.29	1 / 1	jhipster	online-store-app
notification	Unknown	Cluster IP	10.11.248.77	1 / 1	jhipster	online-store-app
notification-mongodb	OK	Cluster IP	None	1 / 1	jhipster	online-store-app
store	OK	Load balancer	34.77.214.210:8080	1 / 1	jhipster	online-store-app
store-mysql	OK	Cluster IP	10.11.242.237	1 / 1	jhipster	online-store-app

The application can be accessed at the external IP we found from the preceding command, for example, `http://34.77.214.210:8080/`:



14. Similarly, you can find the JHipster Console external IP as well using `kubectl get svc jhipster-console -n jhipster` and access it on port 5601.



The JHipster Registry is deployed in headless mode. In order to check the JHipster Registry, we can explicitly expose the service by using this command: `kubectl expose service jhipster-registry --type=LoadBalancer --name=exposed-registry -n jhipster`. Then, we can access the application via the external IP of exposed-registry on port 8761.

15. You can also scale the application by using the following command:

```
> kubectl scale deployment/<app-name> --replicas <number-of-replicas> -n jhipster
```

For example, run `kubectl scale deployment/invoice --replicas=2 -n jhipster`. Now, if you run `kubectl get pods -n jhipster`, you will see that there are two pods for the invoice application. If you check the JHipster Registry, you will see two applications registered for INVOICE:

Instances Registered		
App	Instance ID	Status
INVOICE	invoice:cfeb622b5116d7f33eedc96653c15000	UP
INVOICE	invoice:98a2b43dedac69c14724590146a6d36e	UP
JHIPSTER-REGISTRY	jhipsterRegistry:920af2dc070d7995058495a3de376e27	UP
JHIPSTER-REGISTRY	jhipsterRegistry:894a6b4b44d4bf47434dcef770608446	UP
NOTIFICATION	notification:e27f9108a63a058bbccce3ccd1df22b8	UP
STORE	store:248f98ff9efba6ded30689c09cca3bf9	UP

That is it, we have successfully deployed our entire microservice stack with monitoring and service discovery to Google Cloud using Kubernetes.

Next, we will see how we can use native Kubernetes features for service discovery, monitoring, and so on, using Istio service mesh.

## Using Istio service mesh

JHipster also has support for the Istio service mesh. Let's see how we can create the same microservice architecture using Istio and deploy it to Google Cloud.

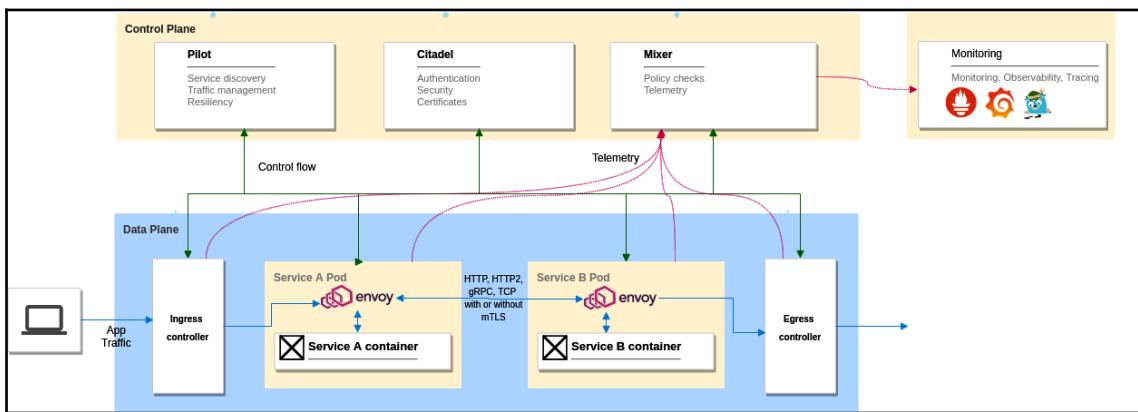
## What is Istio?

Istio (<https://istio.io/docs/concepts/what-is-istio/>) is a service mesh for distributed application architectures, especially the ones with Kubernetes. Istio integrates well with Kubernetes to provide a seamless experience to the end user.

Istio provides the following functionality in a microservice application architecture:

- **Service discovery:** This is similar to Netflix Eureka or Consul
- **Automatic load balancing:** This is similar to Netflix Zuul
- **Routing, circuit breaking, retries, failovers, fault injection:** These are similar to Netflix Ribbon, Hystrix, and so on
- **Policy enforcement for access control, rate limiting, A/B testing, traffic splits, and quotas:** Some features are similar to those in Zuul
- **Metrics, logs, and traces:** These are similar to the JHipster Console
- Secure service-to-service communication

The following is the architecture of Istio:



Istio has two distinct planes:

- **Data plane:** It is made of Envoy (<https://www.envoyproxy.io/>) proxies deployed as sidecars to the application containers. They control all the incoming and outgoing traffic to the container and help with secure interservice communication.
- **Control plane:** It is made up of Pilot, Mixer, Citadel, and Galley. Pilot manages and configures the proxies to route traffic. Mixer enforces policies and collects telemetry. Citadel manages security, while Galley manages configurations.

For monitoring and observability, the default Istio Helm charts can configure an instance of Grafana (<https://grafana.com/>), Prometheus (<https://prometheus.io/>), Jaeger (<https://www.jaegertracing.io/>), and Kiali (<https://www.kiali.io/>). You can use these or use your existing monitoring stack as well if you wish to do so.



Helm (<https://helm.sh/>) is a package manager for Kubernetes. Kubernetes templates are bundled as Helm charts and can be distributed using the Helm repository. This helps with creating reusable Kubernetes manifests.

This is just an overview of Istio; head over to the Istio documentation at <https://istio.io> to learn more about it.

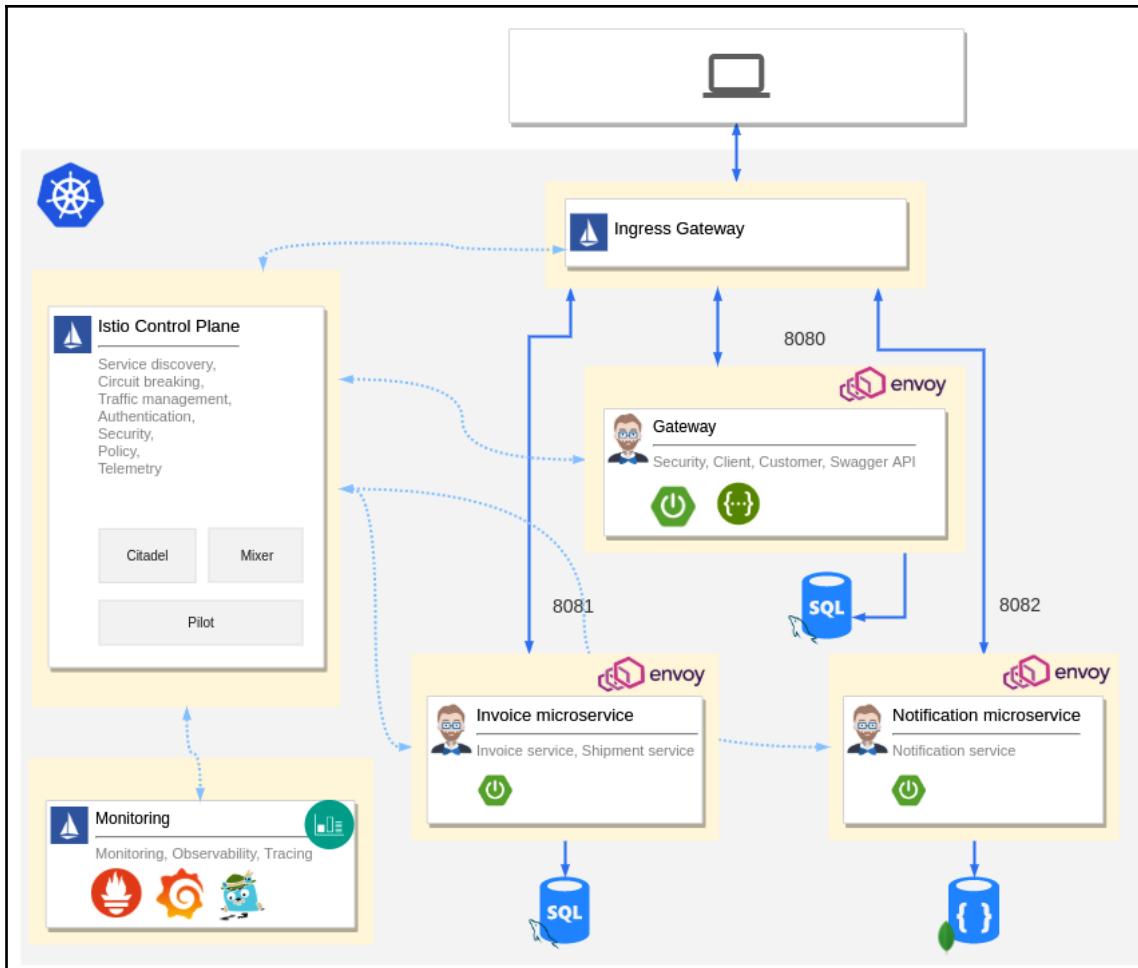
Let's see how we can rebuild the same microservice stack to use Istio.

## Microservice with Istio service mesh

Let's create a new folder for this microservice stack. Let's call this folder `e-commerce-app-istio`. Make sure you are outside our existing `e-commerce-app` folder. Navigate into the folder and copy the JDL we created earlier for our application:

```
> mkdir e-commerce-app-istio && cd e-commerce-app-istio  
> cp ../e-commerce-app/app.jdl app.jdl
```

Now, we need to make some small changes here. First, we need to set `serviceDiscoveryType` to `no` in all the application definitions. This removes the JHipster Registry and Eureka-related code from our applications. We won't be needing these as we will use Istio for that. Our new architecture is as follows:



As you can see, we have replaced JHipster Registry with Istio, and JHipster Console with Istio's monitoring setup. Our applications have the Envoy proxy attached to them as sidecars (meaning it runs in the same pod as our application container). There is a new Ingress Gateway in front of our store gateway. This Ingress is the only entry point into our application and you can see that routing is now handled by the Ingress Gateway and our application gateway is just another microservice.

Next, we can define the deployment option in the JDL. Earlier, we used the Docker Compose and Kubernetes sub-generators, but the same can be defined using the JDL as well.

Similar to other JDL options, it's a simple syntax (you can learn about all the supported options here: <https://www.jhipster.tech/jdl/deployments>):

```
deployment {  
    <deployment option name> <deployment option value>  
}
```

For our new application stack, let's define the following deployment option at the end of the JDL file:

```
/**  
 * Deployments  
 */  
  
deployment {  
    deploymentType kubernetes  
    appsFolders [store, invoice, notification]  
    dockerRepositoryName "deepu105"  
    serviceDiscoveryType no  
    istio true  
    kubernetesServiceType Ingress  
    kubernetesNamespace jhipster  
    ingressDomain "<IngresIp>.nip.io"  
}
```

We have defined a Kubernetes deployment that is similar to what we generated earlier. The only difference is we have set `serviceDiscoveryType` to `no`, set the `kubernetesServiceType` as `Ingress`, and enabled Istio. We also need to set an `ingressDomain`, but we cannot do that yet – let's see why.

When we set the Kubernetes service type as `Ingress`, it would need a valid DNS name to be provided as an Ingress domain. If you already have a registered DNS, you can use that here and map it later to our actual Ingress Gateway that would be created by Istio; otherwise, we can do this only after creating our cluster and deploying Istio to it.

## Deploying Istio to a Kubernetes cluster

Let's create a Kubernetes cluster and install Istio on it.

Make sure you have Helm installed; otherwise, go to <https://helm.sh/docs/intro/install/> and install Helm first. The commands are for Helm v3, as follows:

1. Let's create a Kubernetes cluster on **Google Cloud Platform (GCP)**. Since we already configured the GCP project, we'll use the same and create a cluster directly by executing the following command:

```
> gcloud container clusters create online-store-istio \
    --cluster-version 1.13 \
    --num-nodes 4 \
    --machine-type n1-standard-2
```

The parameters passed are important; we need at least four nodes in the cluster and a bigger CPU. Once the cluster is created, we need to install Istio on it using Helm.

2. Now, let's install Istio locally on our machine. Execute the following commands to fetch Istio:

```
> cd ~/
> export ISTIO_VERSION=1.3.0
> curl -L https://git.io/getLatestIstio | sh -
# Link the installed version to a generic name
> ln -sf istio-$ISTIO_VERSION istio
> export PATH=~/istio/bin:$PATH
```

3. Now, let's create a role binding on our Kubernetes cluster. This is required by Istio:

```
> kubectl create clusterrolebinding cluster-admin-binding \
    --clusterrole=cluster-admin \
    --user="$(gcloud config get-value core/account)"
```

4. Now, we need to create a namespace for Istio so that we can clearly segregate our application from the Istio infrastructure:

```
> kubectl create namespace istio-system
```

5. Now, we can install the Istio **custom resource definitions (CRD)** using Helm:

```
> cd ~/istio-$ISTIO_VERSION

# Install the Istio CRDs
> helm template istio-init install/kubernetes/helm/istio-init --namespace istio-system | kubectl apply -f -
```

6. Verify all the required CRDs are installed. It should output 23 for this version of Istio:

```
> kubectl get crds | grep 'istio.io\|certmanager.k8s.io' | wc -l
```

7. Let's install the Istio components with Helm. We will install the Istio demo setup so that we get Grafana, Jaeger, and Kiali as well. For production, use the Istio default setup. Refer to <https://istio.io/docs/setup/kubernetes/install/helm/> to learn more:

```
> helm template istio install/kubernetes/helm/istio --namespace istio-system \
--values install/kubernetes/helm/istio/values-istio-demo.yaml | kubectl apply -f -
```

8. Wait for the installation to be ready. You can run a watch loop with `watch kubectl get pods -n istio-system`:

Every 2.0s: kubectl get pods -n istio-system					
NAME	READY	STATUS	RESTARTS	AGE	
grafana-54ff8d5c44-hgvhc	1/1	Running	0	27s	
istio-citadel-5778fd44d9-thzcv	0/1	ContainerCreating	0	26s	
istio-cleanups-secrets-1.3.0-l6dj2	0/1	Completed	0	13s	
istio-egressgateway-757668bf5c-qdjqw6	0/1	Running	0	28s	
istio-galley-5596777d77-v5kfx	0/1	ContainerCreating	0	28s	
istio-grafana-post-install-1.3.0-cns2z	0/1	Completed	0	23s	
istio-ingressgateway-74cc7678b6-85rwk	0/1	Running	0	27s	
istio-init-crd-10-1.3.0-9krn8	0/1	Completed	0	5m9s	
istio-init-crd-11-1.3.0-zgqbh	0/1	Completed	0	5m9s	
istio-init-crd-12-1.3.0-xm69t	0/1	Completed	0	5m8s	
istio-pilot-7bbdc4969c-6fgxz	1/2	Running	0	26s	
istio-policy-69867c77c6-w2w24	2/2	Running	0	26s	
istio-security-post-install-1.3.0-j79g5	0/1	Completed	0	23s	
istio-sidecar-injector-5dbb5db74b-kvwkz	0/1	ContainerCreating	0	26s	
istio-telemetry-6f7974dd46-p5zfq	2/2	Running	0	27s	
istio-tracing-66875fddwf5lm	0/1	ContainerCreating	0	26s	
kiali-f7469fb-cfxgx	0/1	Running	0	27s	
prometheus-64464c7dc4-8nrrv	0/1	ContainerCreating	0	26s	

9. Once everything is in **Running** status, we can fetch the external IP of the Istio Ingress Gateway by running this command. Copy the IP, as follows:

```
> kubectl get svc istio-ingressgateway -n istio-system
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
istio-ingressgateway	LoadBalancer	10.7.248.81	34.77.78.2

10. Now, update this IP in the JDL file we created so that `ingressDomain` is "34.77.78.2.nip.io".

You can find the final JDL at <http://bit.ly/jh-book-istio-jdl>.

That is it, we have Istio set up in our Kubernetes cluster. Let's generate our application stack and deploy it.

## Generating the application

Make sure you are in the `e-commerce-app-istio` folder, run `jhipster import-jdl app.jdl`, and wait for the application generation to complete. You will see an output similar to this:

```
WARNING! Kubernetes configuration generated, but no Jib cache found
If you forgot to generate the Docker image for this application, please run:
To generate the missing Docker image(s), please run:
./gradlew bootJar -Pprod jibDockerBuild in /home/deepu/Documents/jhipster-book/v2/e-commerce-app-istio/store
./gradlew bootJar -Pprod jibDockerBuild in /home/deepu/Documents/jhipster-book/v2/e-commerce-app-istio/invoice
./gradlew bootJar -Pprod jibDockerBuild in /home/deepu/Documents/jhipster-book/v2/e-commerce-app-istio/notification

WARNING! You will need to push your image to a registry. If you have not done so, use the following commands to tag and push the images:
docker image tag store deepu105/store
docker push deepu105/store
docker image tag invoice deepu105/invoice
docker push deepu105/invoice
docker image tag notification deepu105/notification
docker push deepu105/notification

INFO! Alternatively, you can use Jib to build and push image directly to a remote registry:
./gradlew bootJar -Pprod jibBuild -Djib.to.image=deepu105/store in /home/deepu/Documents/jhipster-book/v2/e-commerce-app-istio/store
./gradlew bootJar -Pprod jibBuild -Djib.to.image=deepu105/invoice in /home/deepu/Documents/jhipster-book/v2/e-commerce-app-istio/invoice
./gradlew bootJar -Pprod jibBuild -Djib.to.image=deepu105/notification in /home/deepu/Documents/jhipster-book/v2/e-commerce-app-istio/notification

You can deploy all your apps by running the following script:
bash kubectl-apply.sh

Use these commands to find your application's IP addresses:
kubectl get svc store -n jhipster

INFO! Congratulations, JHipster execution is complete!
INFO! Deployment: child process exited with code 0
```

The applications are generated and we have our deployment generated as well. Compared to our previous application, the difference is that there is no JHipster Registry- or JHipster Console-related code in this application. Also, in the generated Kubernetes manifests, we have additional files for Istio-related setup. The additional configs are as follows:

- `*-destination-rule.yml`: Each application will have the Istio destination rule spec, which defines the traffic policy for that application.
- `*-virtual-service.yml`: Each application will have an Istio virtual service spec that is used by the Ingress Gateway to route traffic.
- `store-gateway.yml`: This defines the store app as an entry point for the Ingress Gateway and its routing configuration.
- `istio/grafana-gateway.yml`: This defines a gateway and virtual service for Grafana, which is used for viewing monitoring dashboards.
- `istio/kiali-gateway.yml`: This defines a gateway and virtual service for Kiali, which is used for observability.
- `istio/zipkin-gateway.yml`: This defines a gateway and virtual service for Zipkin, which is used for distributed tracing. We haven't set up Zipkin on our cluster but, with a minor change, we can make this work with Jaeger configured by the Istio demo setup. Follow the step as given:
  - In the generated `kubernetes/istio/zipkin-gateway.yml` file, under the virtual service, change the `http.route.destination.host` value from `zipkin` to `jaeger-query`. This works since Jaeger is backward compatible with Zipkin. You can also go a step further and replace all occurrences of Zipkin in this file with Jaeger if you like, but it's not necessary.

Now, let's deploy everything.

# Deploying to Google Cloud

The console output from JHipster gave us some handy commands for the next steps. Since we have new applications, we need to build them first, as follows:

1. Let's go into each application folder and run the Gradle build. We can use the `jib` command to build and push directly to our registry, in this case, Docker Hub. If you want to build and tag locally, you can use the `jibBuild` command. Change the image name for each application:

```
> ./gradlew bootJar -Pprod jib -Djib.to.image=deepu105/store
```

2. Since these are fresh applications, there should not be any errors. Once the builds are complete, we can navigate into the `kubernetes` folder and execute the provided bash script to start the deployment:

```
> cd kubernetes  
> ./kubectl-apply.sh
```

```
....  
#####  
Please find the below useful endpoints,  
Gateway - http://store.jhipster.34.77.78.2.nip.io  
Zipkin - http://zipkin.istio-system.34.77.78.2.nip.io  
Grafana - http://grafana.istio-system.34.77.78.2.nip.io  
Kiali - http://kiali.istio-system.34.77.78.2.nip.io  
#####
```

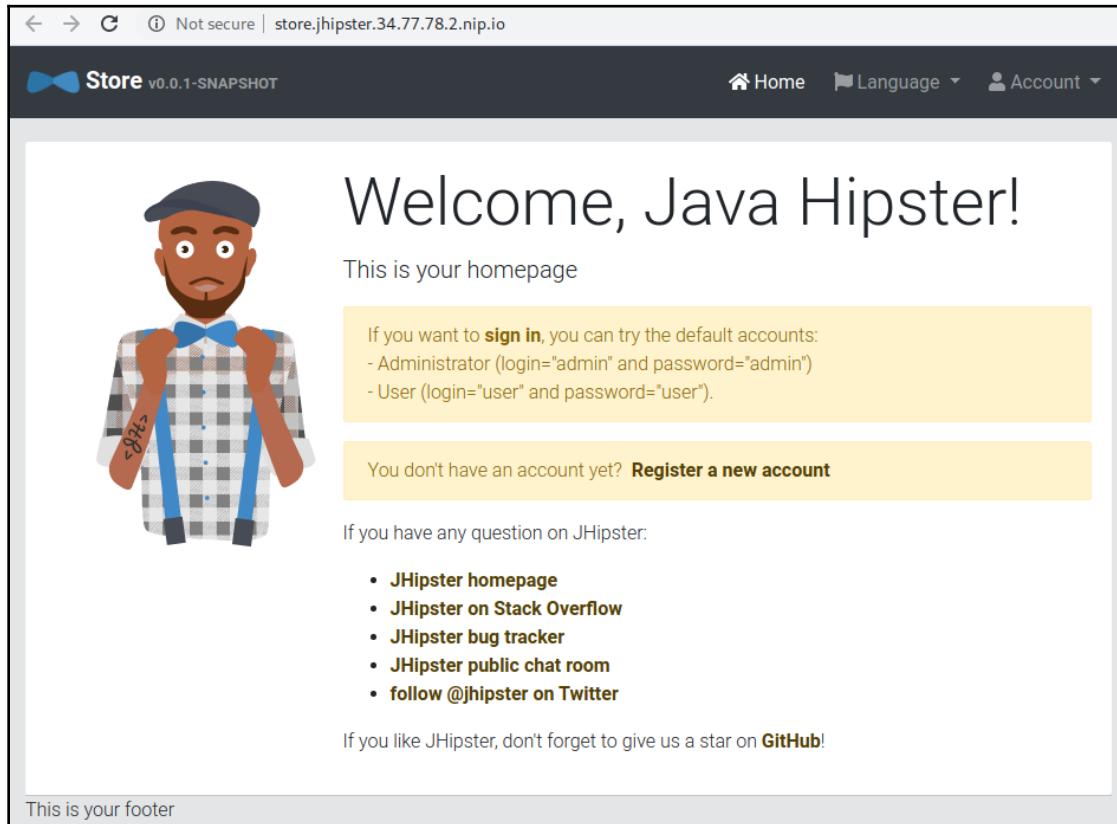
3. This will start the deployment and print out important URLs. We can run a `watch` loop to observe the pod's creation:

```
> watch kubectl get pods -n jhipster
```

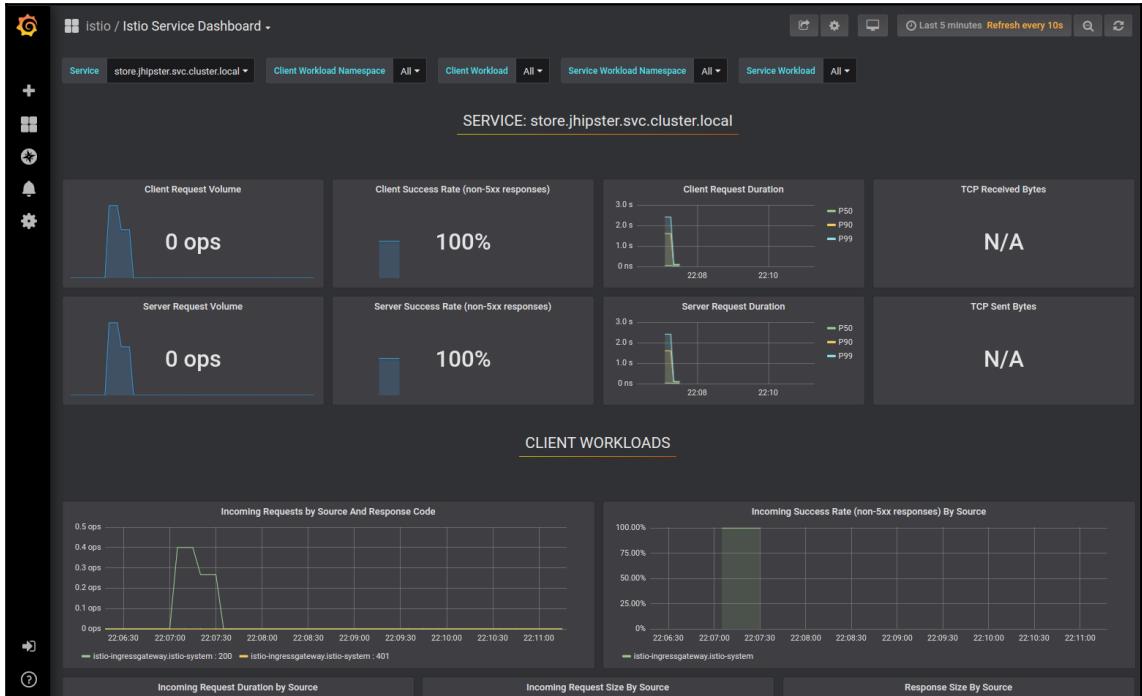
This will give us something similar to the following:

NAME	READY	STATUS	RESTARTS	AGE
invoice-6df4886bc5-xndsm	2/2	Running	0	109s
invoice-mysql-6d87747784-jzgj2	1/1	Running	0	109s
notification-654699dcc6-5p4tl	2/2	Running	0	107s
notification-mongodb-0	1/1	Running	0	107s
store-66bdb5fdb6-zzq26	2/2	Running	0	110s
store-mysql-7fb4f47776-vk4t5	1/1	Running	0	110s

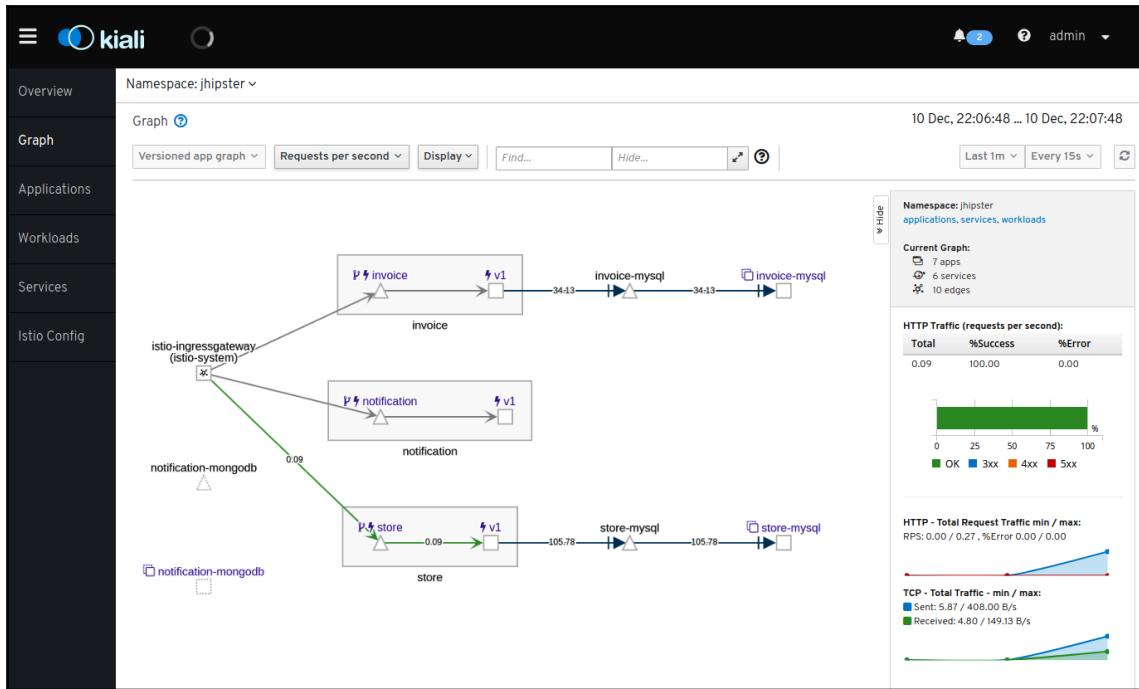
You will notice that we are running fewer items under the JHipster namespace since we are using Istio for service discovery, configurations, and monitoring. Once the pods are in **Running** status, we can visit the application URL, for example, <http://store.jhipster.34.77.78.2.nip.io/>, to see our application in action:



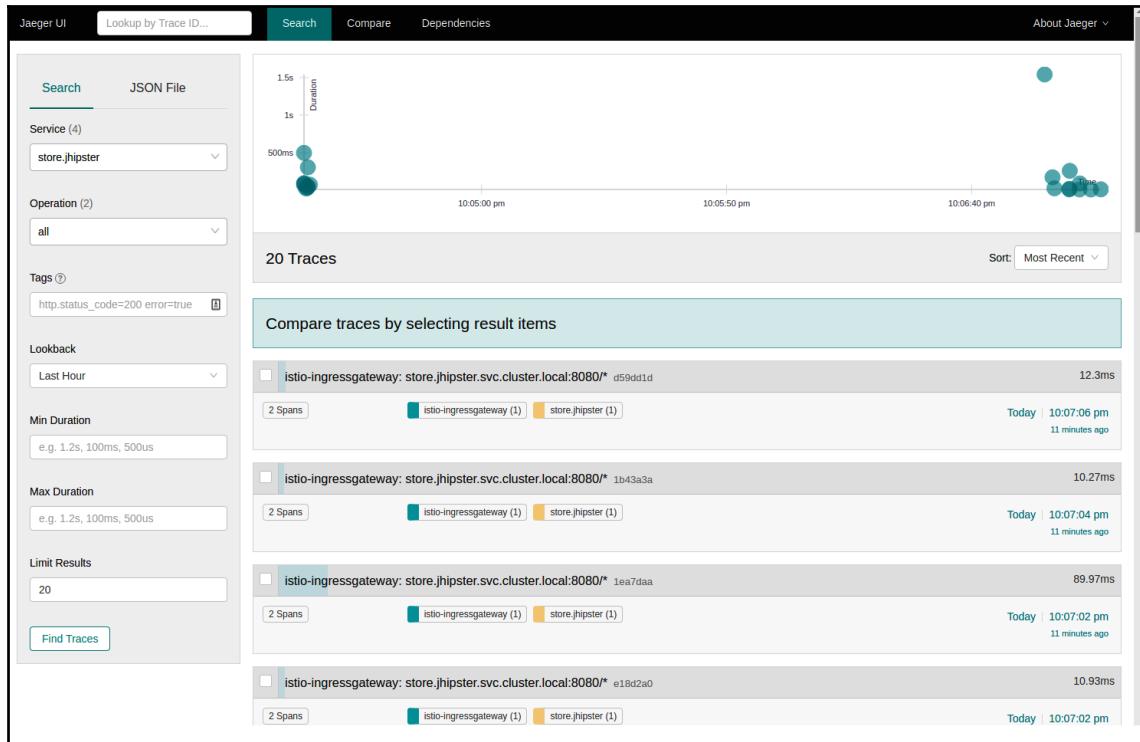
You can also visit Grafana and Kiali endpoints (log in using your username/password admin/admin) to see the metrics at their corresponding URLs, printed at the end of the application generation process by JHipster. Here is one of the Grafana dashboards:



Here is one of the Kiali graphs showing how services are connected:



For Jaeger, visit the URL printed for Zipkin as we reconfigured it to route to Jaeger:



This is showing traces for the store application. You can click on a trace to dig deeper.

## Summary

Orchestrating your containers is the most difficult task to perform in a microservices environment. Kubernetes, as a container orchestrator, stands out in solving this. We have seen how to generate the configuration file for Kubernetes with JHipster, followed by deploying the application to Google Cloud. We also saw how to use the Istio service mesh with our applications.

So far, we've seen how we can develop and deploy an e-commerce application using JHipster. We started with a monolith and we successfully scaled it into a microservice architecture. We finally deployed it with all the bells and whistles to a cloud service provider using Kubernetes—all with the help of JHipster and the various tools and technologies it supports. With this chapter, our journey of developing the e-commerce web application comes to an end and we hope you had a wonderful experience following it through.

In the next chapter, we will see how we can use JHipster further to create an application with a React or Vue.js client-side, so stay tuned.

# 6

## Section 6: React and Vue.js for the Client Side

In the first chapter of this section, instead of generating a JHipster application with Angular, you will generate an application with React on the client side. The following chapter does the same, but this time, Vue.js will be used as the client-side framework. In the final chapter of this section, an entire summary of all the knowledge acquired throughout this book is presented, along with some best practices and next steps to optimize your skills.

This section comprises the following chapters:

- Chapter 13, *Using React for the Client-Side*
- Chapter 14, *Using Vue.js for the Client-Side*
- Chapter 15, *Best Practices with JHipster*

# 13

## Using React for the Client-Side

So far, we have learned how to build web applications and microservices with Angular as the client-side framework. AngularJS was the most popular client-side framework until the new Angular framework was released. Angular caused major disruptions due to its backward-incompatible architecture and gave way to more people migrating to React. Hence, the tides have shifted, and now React is the most popular and sought-after client-side framework, followed by Vue.js and Angular. JHipster has first-class support for React as well and can be used instead of Angular if you prefer. In this chapter, you will learn about the architecture that's used to build **React** apps in JHipster and how to use React as the client-side framework.

In this chapter, we will cover the following topics:

- Generating an application with React client-side
- Technical stack and source code
- Generating an entity with React client-side

### Generating an application with React client-side

In this section, we'll create a React application with JHipster. You will need to open a Terminal to run the following commands:

1. Create a new folder and navigate to it by running `mkdir jhipster-react && cd jhipster-react`.
2. Now, run the `jhipster` command in the Terminal.

3. JHipster will start with prompts; let's select default options for everything except for the question **Which \*Framework\* would you like to use for the client?** For this question, select **React** as the value and proceed.
4. Once all the prompts have been completed, JHipster will generate the application and start installing dependencies before starting the webpack build.



You can run `npm run prettier:format` to format the client-side code anytime. It will also be automatically run whenever you commit something with a Git pre-commit hook.

Our selected options will look as follows:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? jhreact
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? MySQL
? Which *development* database would you like to use? H2 with disk-based persistence
? Do you want to use the Spring cache abstraction? Yes, with the Ehcache implementation (local cache, for a single node)
? Do you want to use Hibernate 2nd level cache? Yes
? Would you like to use Maven or Gradle for building the backend? Maven
? Which other technologies would you like to use? (Press <space> to select, <a> to toggle all, <i> to invert selection)
? Which *Framework* would you like to use for the client? React
? Would you like to use a Bootswatch theme (https://bootswatch.com/)?
Default JHipster
? Would you like to enable internationalization support? Yes
? Please choose the native language of the application English
? Please choose additional languages to install (Press <space> to select, <a> to toggle all, <i> to invert selection)
? Besides JUnit and Jest, which testing frameworks would you like to use? (Press <space> to select, <a> to toggle all, <i> to invert selection)
? Would you like to install other generators from the JHipster Marketplace? No
```

That's it; we're done. Our first JHipster React application was created successfully. Now, let's start the application so that we can play around.

Since we had chosen the default **Maven build** option, JHipster created a wrapper for it, so let's start our server by running `./mvnw` in a Terminal.



You could choose Gradle instead of Maven if you prefer, just like we've done in previous chapters. It is good to mix it up a little to get some exposure to different technologies.

Maven will download the necessary dependencies and start the Spring Boot application using the embedded Undertow container. Once the application starts successfully, we will see the following in the console:

```
2019-12-10 23:11:29.855 INFO 1043043 --- [ restartedMain]
com.mycompany.store.JhreactApp :

-----
Application 'jhreact' is running! Access URLs:
Local: http://localhost:8080/
External: http://192.168.2.177:8080/
Profile(s): [dev, swagger]
```

Visit the URL (`http://localhost:8080`) in your favorite browser to see the following application in action:

The screenshot shows the JHipster homepage with the following content:

- Welcome, Java Hipster!**
- This is your homepage
- If you want to **sign in**, you can try the default accounts:
  - Administrator (login="admin" and password="admin")
  - User (login="user" and password="user").
- You don't have an account yet? **Register a new account**
- If you have any question on JHipster:
  - [JHipster homepage](#)
  - [JHipster on Stack Overflow](#)
  - [JHipster bug tracker](#)
  - [JHipster public chat room](#)
  - [follow @jhipster on Twitter](#)
- If you like JHipster, don't forget to give us a star on [Github!](#)
- This is your footer

You will see the home screen with the hipster person looking back at you (see the preceding screenshot). The only difference you will notice here is that the image is on the right-hand side instead of the left. Other than that, every screen should look identical to the one in Angular.

Go ahead and log in using the default admin user and play around.

The application looks exactly the same as the Angular application we built earlier, except for the image, of course, and has all the same account and administration modules.

This will make things more interesting when we look at the technical stack and source code.

## Technical stack and source code

Before we dive into the generated code, let's talk about the technical stack. We looked at React in Chapter 2, *Getting Started with JHipster*, but let's recap.

React is a view rendering library that was created by Jordan Walke in 2011, and was open sourced in May 2013. It is maintained and backed by Facebook and has a huge community behind it. React follows the JS in HTML approach, where the markup code is written using JavaScript. To reduce verbosity, React uses a syntax sugar for JavaScript called JSX (<https://reactjs.org/docs/introducing-jsx.html>) to describe **view** elements. It looks similar to HTML, but it is not exactly HTML as some of the standard HTML attributes, such as class, for example, are renamed to *className*, and attribute names are written using *camelCase* rather than dash-case.

For example, the following is a JSX snippet. You always have to use React in context for JSX to work:

```
const element = <div><strong>Hello there</strong></div>
```

When it comes to TypeScript, the JSX extension becomes TSX.

React uses a concept called virtual DOM to improve rendering efficiency. Virtual DOM is a lightweight copy of the actual DOM, and by comparing the virtual DOM after an update against the virtual DOM snapshot before the update, React can decide what exactly changed and render only that on to the actual DOM, hence making render cycles fast and efficient.

React components can have their own state and you can pass various properties to a component, all of which are available to the component as props.

Unlike Angular, React is not a full-fledged MVVM framework. It is just a view rendering library and, hence, when building React applications, we would always have to add a few more libraries for things such as state management, and routing.

## Technical stacks

The following are the technical stacks that are used by JHipster when React is chosen as the client-side framework:

- **Rendering:** React written using TypeScript
- **State management:** Redux + React Redux + Redux Promise Middleware + Redux Thunk
- **Routing:** React Router
- **HTTP:** Axios
- **Responsive design:** Bootstrap 4 + reactstrap
- **Linting:** ESLint + TSLint
- **Utilities:** Lodash
- **Unit testing:** Jest + Enzyme
- **Build:** Webpack

Let's look at some of the most important components of the stack.

## Using TypeScript

The client-side is built using React, but instead of going with the traditional JavaScript ES6, we are using TypeScript as the language of choice.



Visit <http://bit.ly/react-typescript> to learn how TypeScript makes React components nicer. Also, visit <https://github.com/piotrwritek/react-redux-typescript-guide> to learn about how to make the most out of TypeScript + React.

This gives you the flexibility to use some of the concepts that you may already be familiar with if you come from a server-side background. It also provides static type checking, which makes development more efficient and less error-prone.

## State management with Redux and friends

React provides basic state management within React components, but sometimes, this is not sufficient, especially when your application needs to share state between multiple components. State management solutions such as Flux, Redux, and MobX are quite popular in the React world and JHipster uses Redux as the state management layer.

When should you use the React component state?



- **If the variable can always be calculated using a prop:** Don't use the component state; calculate the variable during rendering
- **If the variable is not used in rendering but to hold data:** Don't use the component state; use private class fields
- **If the variable is obtained from an API and is required by more than one component:** Don't use the component state; use the Redux global state and pass the variable as a prop

Redux (<https://redux.js.org/>) is a predictable state management solution for JavaScript that evolved from the Flux concept (<https://facebook.github.io/flux/>). Redux provides a global immutable store that can only be updated by emitting or dispatching actions. An action is an object that describes what changed, and it uses a pure reducer function to transform the state. A reducer is a pure function (a function that does not cause any side effects) that takes in the current state and an action. It returns a new state after applying any logic that's specified in the function.

React Redux is a binding for Redux that provides a higher-order component called `connect` for React, which is used to connect React components to the Redux store. For example, let's take a look at `src/main/webapp/app/modules/login/login.tsx`:

```
export const Login = (props: ILoginProps) => {
  const [showModal, setShowModal] = useState(props.showModal);

  useEffect(() => {
    setShowModal(true);
  }, []);

  ...

};

const mapStateToProps = ({ authentication }: IRootState) => ({
  isAuthenticated: authentication.isAuthenticated,
  loginError: authentication.loginError,
```

```
        showModal: authentication.showModalLogin
    });

const mapDispatchToProps = { login };

type StateProps = ReturnType<typeof mapStateToProps>;
type DispatchProps = typeof mapDispatchToProps;

export default connect(mapStateToProps, mapDispatchToProps)(Login);
```

In this component, we have the following:

- The `mapStateToProps` function is used to map properties from the global Redux store to the component's props.
- The `mapDispatchToProps` function is used to wrap the given functions with the Redux dispatch call.
- The `useState` hook provided by React lets us use a local state without writing a TypeScript class.
- The `useEffect` hook provided by React lets us perform side effects in function components.

Redux Promise Middleware (<https://github.com/pburtchaell/redux-promise-middleware>) is used to handle asynchronous action payloads. It accepts a Promise and dispatches pending, fulfilled, and rejected actions based on the Promise state. It is useful when Redux actions are making HTTP requests or performing async operations.

Redux Thunk (<https://github.com/gaearon/redux-thunk>) is another piece of middleware that's used to chain actions. It is useful when an action has to call another action based on certain conditions or in general to handle side effects.

## Routing with React Router

React Router (<https://reacttraining.com/react-router/web/guides/philosophy>) is used for client-side routing. The default setup with JHipster is to use browser history-based routing (HTML5 push state). It provides simple component-based routing, along with a flexible API for advanced routing setups. Routes can be defined anywhere in the application alongside the normal React rendering code. JHipster provides some custom wrappers such as `PrivateRoute` to enable authorization-based routing and `ErrorBoundaryRoute` with custom React error boundaries. JHipster also does lazy loading for entities and admin and account screens.

Let's take a look at `src/main/webapp/app/routes.tsx`:

```
...
// Lazy loading for account screens
const Account = Loadable({
  loader: () => import(/* webpackChunkName: "account" */ 
'./app/modules/account'),
  loading: () => <div>loading ...</div>
});
// Lazy loading for admin screens
const Admin = Loadable({
  loader: () => import(/* webpackChunkName: "administration" */ 
'./app/modules/administration'),
  loading: () => <div>loading ...</div>
});

const Routes = () => (
  <div className="view-routes">
    <Switch>
      <ErrorBoundaryRoute path="/login" component={Login} />
      ...
      <ErrorBoundaryRoute path="/account/reset/finish/:key?" 
        component={PasswordResetFinish} />
      <PrivateRoute path="/admin" component={Admin} 
        hasAnyAuthorities={[AUTHORITIES.ADMIN]} />
      <PrivateRoute path="/account" component={Account} 
        hasAnyAuthorities={[AUTHORITIES.ADMIN, AUTHORITIES.USER]} />
      <ErrorBoundaryRoute path="/" exact component={Home} />
        <PrivateRoute path="/" component={Entities} 
          hasAnyAuthorities={[AUTHORITIES.USER]} />
      <ErrorBoundaryRoute component={PageNotFound} />
    </Switch>
  </div>
);
export default Routes;
```

The parent routes of our application are defined here; the child routes will be defined in their corresponding modules.

## HTTP requests using Axios

Axios (<https://github.com/axios/axios>) is a Promise-based HTTP client. It is a powerful and flexible library with a very straightforward API. It is used to fetch data from the JHipster application's server-side REST endpoints from Redux actions. The resulting Promise is resolved by the Redux Promise Middleware to provide data to the reducer.

The following code shows a Redux action with an asynchronous payload:

```
export const getEnv = () => ({
  type: ACTION_TYPES.FETCH_ENV,
  payload: axios.get('management/env')
});
```

Axios is used to fetch the payload and return the promise that will be resolved by our Redux middleware.

## Bootstrap components using reactstrap

JHipster uses Bootstrap 4 as its UI framework. Since we are building a React application, it makes sense to use a Native React binding instead of Bootstrap's jQuery-based components. Reactstrap (<https://reactstrap.github.io/>) provides pure React components for Bootstrap 4. We also make use of the Availability reactstrap Validation (<https://availability.github.io/availability-reactstrap-validation/>) library, which provides form validation support for reactstrap form elements.

Let's take a look at `src/main/webapp/app/modules/login/login-modal.tsx`:

```
<Modal isOpen={this.props.showModal} toggle={handleClose} backdrop="static"
  id="login-page" autoFocus={false}>
  <AvForm onSubmit={this.handleSubmit}>
    <ModalHeader id="login-title" toggle={handleClose}>
      <Translate contentKey="login.title">Sign in</Translate>
    </ModalHeader>
    <ModalBody>
      <Row>
        <Col md="12">
          ...
        </Col>
      </Row>
      ...
    </ModalBody>
    <ModalFooter>
      <Button color="secondary" onClick={handleClose} tabIndex="1">
        <Translate contentKey="entity.action.cancel">Cancel</Translate>
      </Button>
    </ModalFooter>
  </AvForm>
</Modal>
```

```
</Button>{' '}<br/>
<Button color="primary" type="submit">
  <Translate contentKey="login.form.button">Sign in</Translate>
</Button>
</ModalFooter>
</AvForm>
</Modal>
```

This a simple component that uses some reactstrap components to build the modal pop-up UI.

## Unit testing setup

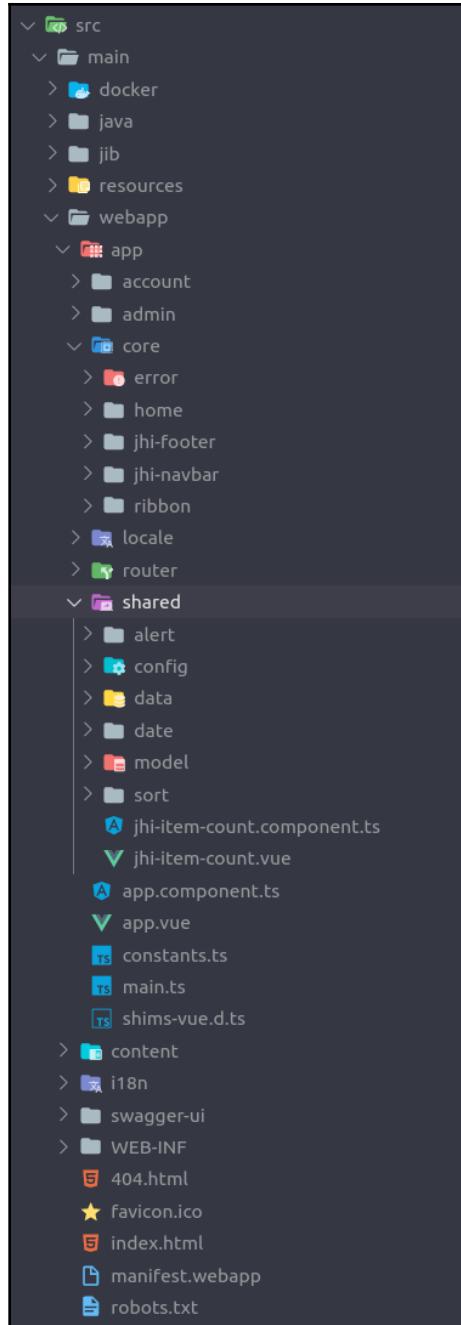
JHipster uses a combination of Jest and Enzyme to unit test the client-side components.

Jest (<https://jestjs.io/>) is used as the testing framework. Enzyme (<http://airbnb.io/enzyme/>) is a testing utility for React that makes it easy to unit test React components. In combination, these libraries provide a rich and intuitive testing environment for React.

Let's run the generated unit tests. Run `npm test` in a Terminal. The tests should all pass out of the box.

## Generated source code

Let's take a look at the generated code. Since we looked at the server-side code in the previous chapters, we will only look at the client-side code here:



The structure is quite similar to what we saw for Angular, but the React code is organized slightly differently. Here, we are only concerned with the code inside `src/main/webapp/app` since everything else is exactly the same as what we saw for the Angular application.

Let's take a look at some of the important parts of the code:

- `index.tsx`: This is the entry point of our application. This is where we Bootstrap React to the `root` `div` and initialize the Redux store:

```
...
const store = initStore();
registerLocale(store);

const actions = bindActionCreators({ clearAuthentication },
  store.dispatch);
setupAxiosInterceptors(() =>
  actions.clearAuthentication('login.error.unauthorized'));
...

const rootEl = document.getElementById('root');

const render = Component =>
  ReactDOM.render(
    <ErrorBoundary>
      <Provider store={store}>
        <div>
          ...
          <Component />
        </div>
      </Provider>
    </ErrorBoundary>,
    rootEl
  );
  render(AppComponent);
```

- `app.tsx`: This is our main application component. We declare the React Router and the main application UI structure here:

```
...
export const App = (props: IAppProps) => {
  ...
  return (
    <Router basename={baseHref}>
      <div className="app-container" style={{ paddingTop }}>
        <ToastContainer ... />
      <ErrorBoundary>
```

```
<Header
  ...
  />
</ErrorBoundary>
<div className="container-fluid view-container"
  id="app-view-container">
  <Card className="jh-card">
    <ErrorBoundary>
      <AppRoutes />
    </ErrorBoundary>
  </Card>
  <Footer />
</div>
</div>
</Router>
);
};

...
...
```

- `routes.tsx`: This is where the application's main parent routes are defined. They are imported into the `app.tsx` file from here. We looked at this component earlier.
- `config`: This is where framework-level configurations are done. Some important ones are as follows:
  - `axios-interceptor.ts`: HTTP interceptors are configured here. This is where the JWT tokens are set to requests and errors are handled.
  - `constants.ts`: Application constants.
  - `*-middleware.ts`: The error, notification, and logging middleware for Redux is configured here.
  - `store.ts`: Redux store configuration is done here. Middlewares are registered during this stage. The order of the middlewares in the array is important as they act as a pipeline, passing actions from one middleware to another, as shown here:

```
const defaultMiddlewares = [
  thunkMiddleware,
  errorMiddleware,
  notificationMiddleware,
  promiseMiddleware(),
  loadingBarMiddleware(),
  loggerMiddleware
];
```

- `translation.ts`: i18n-related configurations are done here.
- `entities`: The entity modules are present here.
- `modules`: The application UI modules are here:
  - `account`: Account pages such as settings and password reset can be found here.
  - `administration`: The admin screens, such as metric, health, and user management, are here.
  - `home`: Home screen of the application.
  - `login`: Login and logout components.
- `shared`: Shared components and reducers:
  - `auth`: `private-route.tsx`: This is used for authenticated routes.
  - `error`: Custom error boundaries used in the application.
  - `layout`: Layout-related components such as header, footer, and menu.
  - `model`: TypeScript model for entities.
  - `reducers`: Shared reducers used by the application:
    - `authentication.ts`: This is used for authentication-related actions and reducers. Let's take the `LOGIN` action as an example. This action accepts username, password, and remember me and dispatches `ACTION_TYPES.LOGIN` with an asynchronous payload from an HTTP call to authenticate our credentials. We use the `async/await` feature from ES7 to avoid complex callbacks here. The result from the dispatch is obtained when we extract the JWT `bearerToken` and store it in the local or session storage of the browser, based on the remember me setting that's passed. Dispatching `ACTION_TYPES.LOGIN` will trigger the appropriate case in the reducer based on the status of the Promise:

```
...  
  
export const ACTION_TYPES = {  
  LOGIN: 'authentication/LOGIN',  
  ...  
};
```

```
const initialState = {
  ...
};

// Reducer
export default (state = initialState, action) => {
  switch (action.type) {
    case REQUEST(ACTION_TYPES.LOGIN):
    case REQUEST(ACTION_TYPES.GET_SESSION):
      return {
        ...state,
        loading: true
      };
    case FAILURE(ACTION_TYPES.LOGIN):
      return {
        ...initialState,
        errorMessage: action.payload,
        showModalLogin: true,
        loginError: true
      };
    ...
    case SUCCESS(ACTION_TYPES.LOGIN):
      return {
        ...state,
        loading: false,
        loginError: false,
        showModalLogin: false,
        loginSuccess: true
      };
    ...
    default:
      return state;
  }
};
...
export const login =
  (username, password, rememberMe = false) => async
(dispatch, getState) => {
  const result = await dispatch({
    type: ACTION_TYPES.LOGIN,
    payload: axios.post('/api/authenticate', {
      username, password, rememberMe
    })
  });
  const bearerToken =
    result.value.headers.authorization;
  if (bearerToken && bearerToken.slice(0, 7) ===
  'Bearer ') {
    const jwt = bearerToken.slice(7,
```

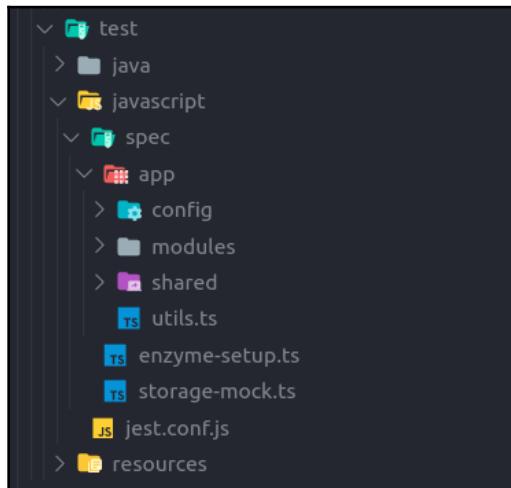
```
bearerToken.length);
    if (rememberMe) {
        Storage.local.set(AUTH_TOKEN_KEY, jwt);
    } else {
        Storage.session.set(AUTH_TOKEN_KEY, jwt);
    }
}
await dispatch(getSession());
};

...

```

- `util`: Utility functions used in the application.

The folder structure of the unit test code is also quite similar to the main `src` folder:



## Generating an entity with React client-side

Now, let's look at how we can create an entity using the JHipster entity generator with a React client-side. We will create a simple employee entity with the name, age, and date of birth fields:

1. Open a Terminal, navigate to the folder of the React app, and run `jhipster entity employee`.

2. Create the fields one by one, select **Yes** for the question **Do you want to add a field to your entity?**, and start filling in the field with **name** for the next question, **What is the name of your field?**
3. Select **String** as the field type for the next question, **What is the type of your field?**
4. For the question **Which validation rules do you want to add?**, choose **Required** for the name field and proceed.
5. Continue this process for the `age` and `dob` fields. `age` is of the integer type, while `dob` is of the instant type.
6. When asked again, **Do you want to add a field to your entity?**, choose **No**.
7. For the next question, **Do you want to add a relationship to another entity?**, choose yes.
8. Provide `user` as the name of the other entity, and as the name of the relationship for the following questions.
9. For the next question, **What is the type of the relationship?**, we'll create a one-to-one relationship with the user.
10. Choose no for the next two questions and no again when asked to add another relationship.
11. For the questions that follow, select the default options and proceed.

The `jhipster entity employee` command will produce the following console output:

```
Using JHipster version installed globally
Executing jhipster:entity employee
Options:

The entity employee is being created.

...
=====
Fields
name (String) required
age (Integer)
dob (Instant)

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

? Do you want to use separate service class for your business

logic? **No**, the REST controller should use the repository directly  
? Is this entity read-only? **No**  
? Do you want pagination on your entity? **No**

Everything is configured, generating the entity...

...

JHipster will generate the entity and run the webpack build.



As an exercise, why don't you try to use the JDL model from the monolith application we built earlier here to generate entities?

12. If your server is not running, start it in a Terminal by running `./mvnw`. If it is already running, then just compile the new code by running `./mvnw compile`; Spring DevTools will restart the app automatically.
13. Start BrowserSync in another Terminal by running `npm start` and check the employee entity we just created:

### Create or edit a Employee

Name

Age

Dob

User

CancelSave

14. Create an entity to check whether everything works fine:

Employees					<a href="#">+ Create a new Employee</a>
ID	Name	Age	Dob	User	
1	mobile Fish	77969	Dec 12, 2019, 1:48 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
2	Table Computer	5808	Dec 12, 2019, 5:09 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
3	seamless Metal deposit	5641	Dec 11, 2019, 4:53 PM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
4	parsing red Chips	18340	Dec 12, 2019, 1:21 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
5	Home Loan Account	706	Dec 12, 2019, 5:48 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
6	intangible	61185	Dec 12, 2019, 6:34 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
7	Gorgeous Frozen Pants	23277	Dec 12, 2019, 5:13 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
8	bluetooth	19967	Dec 12, 2019, 6:59 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
9	Planner Canyon	85994	Dec 12, 2019, 1:35 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
10	Table Small Frozen Sausages Planner	80839	Dec 12, 2019, 6:55 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
11	Superman	30	Jan 1, 1985, 12:00 AM	admin	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>

Now, let's see what happens once the entity's been created:

- For the entity we created, JHipster generated/updated the following files:

```
info Using blueprint generator-jhipster-vuejs for entity-client subgenerator
create .jhipster/Employee.json
create src/main/resources/config/liquibase/changelog/20191212151024_added_entity_Employee.xml
create src/main/resources/config/liquibase/changelog/20191212151024_added_entity_constraints_Employee.xml
create src/main/resources/config/liquibase/fake-data/employee.csv
create src/main/java/com/mycompany/store/domain/Employee.java
create src/main/java/com/mycompany/store/repository/EmployeeRepository.java
create src/main/java/com/mycompany/store/web/rest/EmployeeResource.java
create src/test/java/com/mycompany/store/web/rest/EmployeeResourceIT.java
create src/test/java/com/mycompany/store/domain/EmployeeTest.java
conflict src/main/resources/config/liquibase/master.xml
? Overwrite src/main/resources/config/Liquibase/master.xml? overwrite this and all others
  force src/main/resources/config/liquibase/master.xml
  force src/main/java/com/mycompany/store/config/CacheConfiguration.java
create src/main/webapp/app/entities/employee/employee-details.vue
create src/main/webapp/app/entities/employee/employee.vue
create src/main/webapp/app/entities/employee/employee-update.vue
  force src/main/webapp/app/core/jhi-navbar/jhi-navbar.vue
create src/main/webapp/app/entities/employee/employee-details.component.ts
create src/main/webapp/app/entities/employee/employee.component.ts
create src/main/webapp/app/entities/employee/employee.service.ts
create src/main/webapp/shared/model/employee.model.ts
create src/main/webapp/app/entities/employee/employee-update.component.ts
create src/test/javascript/spec/app/entities/employee/employee.component.spec.ts
create src/test/javascript/spec/app/entities/employee/employee-details.component.spec.ts
create src/test/javascript/spec/app/entities/employee/employee.service.spec.ts
create src/test/javascript/spec/app/entities/employee/employee-update.component.spec.ts
create src/test/javascript/e2e/entities/employee/employee.page-object.ts
create src/test/javascript/e2e/entities/employee/employee.spec.ts
create src/test/javascript/e2e/entities/employee/employee-details.page-object.ts
create src/test/javascript/e2e/entities/employee/employee-update.page-object.ts
  force src/main/webapp/app/router/index.ts
  force src/main/webapp/app/main.ts
create src/main/webapp/i18n/en/employee.json
  force src/main/webapp/i18n/en/global.json
create src/main/webapp/i18n/hi/employee.json
  force src/main/webapp/i18n/hi/global.json
```

- On the React client-side, we have the following files:

```
src/main/webapp/app/entities/employee/employee-detail.tsx
src/main/webapp/app/entities/employee/employee.tsx
src/main/webapp/app/entities/employee/employee.reducer.ts
src/main/webapp/app/shared/model/employee.model.ts
src/main/webapp/app/entities/employee/index.tsx
src/main/webapp/app/entities/employee/employee-delete-dialog.tsx
src/main/webapp/app/entities/employee/employee-update.tsx
src/test/javascript/spec/app/entities/employee/employee-
    reducer.spec.ts
```

- The `index.tsx` file declares the routes for the entity:

```
<Switch>
  <ErrorBoundaryRoute exact path={`${match.url}/:id/delete`} component={EmployeeDeleteDialog} />
  <ErrorBoundaryRoute exact path={`${match.url}/new`} component={EmployeeUpdate} />
  <ErrorBoundaryRoute exact path={`${match.url}/:id/edit`} component={EmployeeUpdate} />
  <ErrorBoundaryRoute exact path={`${match.url}/:id`} component={EmployeeDetail} />
  <ErrorBoundaryRoute path={match.url} component={Employee} />
</Switch>
```

- The `employee.reducer.ts` file declares the actions and reducer for the entity; for example, let's look at the action and reducer that are used to create an entity. The `createEntity` action dispatches `ACTION_TYPES.CREATE_EMPLOYEE` with the HTTP payload. Once the HTTP request resolves, we dispatch the `getEntities` action to fetch the updated entity list. The reducer is common for create and update actions. Let's take a look at the create action and reducer:

```
...
export const ACTION_TYPES = {
  ...
  CREATE_EMPLOYEE: 'employee/CREATE_EMPLOYEE',
  ...
};

const initialState = {
  ...
};

// Reducer
export default (state: EmployeeState = initialState, action): EmployeeState => {
  switch (action.type) {
    ...
    case REQUEST(ACTION_TYPES.CREATE_EMPLOYEE):
      ...
      return {
        ...
      };
    ...
    case FAILURE(ACTION_TYPES.CREATE_EMPLOYEE):
      ...
      return {
```

```

    ...
};

...
case SUCCESS(ACTION_TYPES.CREATE_EMPLOYEE):
case SUCCESS(ACTION_TYPES.UPDATE_EMPLOYEE):
    return {
        ...
    };
...
default:
    return state;
}
};

const apiUrl = 'api/employees';
...

export const createEntity: ICrudPutAction<IEmployee> = entity =>
async dispatch => {
    const result = await dispatch({
        type: ACTION_TYPES.CREATE_EMPLOYEE,
        payload: axios.post(apiUrl, cleanEntity(entity))
    });
    dispatch(getEntities());
    return result;
};
...

```

- employee.tsx, employee-update.tsx, employee-detail.tsx, and employee-delete-dialog.tsx declare the entity listing, entity update, entity detail, and entity delete dialog, respectively. Let's look at employee.tsx. Here, we define the type for the props using a TypeScript interface called **IEmployeeProps**. We trigger the actions to fetch entities when our component mounts using the `useEffect` hook method. The `render` method returns the JSX for the UI. The `Employee` component is connected to the Redux store using the higher-order component. Let's take a look:

```

...
export interface IEmployeeProps extends StateProps, DispatchProps,
RouteComponentProps<{ url: string }> {}

export const Employee = (props: IEmployeeProps) => {
    useEffect(() => {
        props.getEntities();
    }, []);
}

const { employeeList, match } = props;

```

```
        return (
          <div>
            ...
          </div>
        );
      };

      const mapStateToProps = ({ employee }: IRootState) => ({
        employeeList: employee.entities
      });

      const mapDispatchToProps = {
        getEntities
      };

      type StateProps = ReturnType<typeof mapStateToProps>;
      type DispatchProps = typeof mapDispatchToProps;

      export default connect(
        mapStateToProps,
        mapDispatchToProps
      )(Employee);
    
```

The other components also follow a similar approach. Code wise, React code has much less boilerplate and is more concise compared to Angular.

## Summary

In this chapter, we learned some general concepts about React, Redux, and other libraries on the React ecosystem. We also learned how to create a React app using JHipster and generated entities for it. We saw how we can make use of TypeScript with React and also walked through the generated code. Finally, we ran and tested our created application.

In the next chapter, we will learn how to create applications with Vue.js as the client-side framework.

# 14

## Using Vue.js for the Client-Side

So far, we have learned how to build web applications and microservices with Angular and React as the client-side framework. Vue.js is one of the popular client-side frameworks that has been gaining momentum in recent times. It has a great community and is completely driven by its community, unlike Angular and React, which are backed by corporations.

JHipster supports using Vue.js as the client-side by providing an official blueprint. In this chapter, we will learn how to use the official Vue.js module and about the technical stack that's used for Vue.js.



Blueprints are plugins for JHipster that help replace a specific part of the code that's generated for the application.

In this chapter, we will cover the following topics:

- Installing and using a JHipster Blueprint
- Generating an application with Vue.js client-side
- Technical stack and source code
- Generating an entity with Vue.js client-side

## Generating an application with Vue.js client-side

In this section, we'll learn how to install the official JHipster blueprint for Vue.js. You will need to open the Terminal application to run the following commands:

1. Run `npm install -g generator-jhipster-vuejs` on a Terminal to install it. Once the installation is complete, we can create a Vue.js application with JHipster using the blueprint.

2. Create a new folder and navigate to it by running `mkdir jhipster-vuejs && cd jhipster-vuejs`.
3. Now, run the `jhipster --blueprints vuejs` command in the Terminal. The `--blueprints` flag is used to specify the blueprints that JHipster needs to use.
4. JHipster will start with some prompts; let's select the default options for everything except for the build tool and test frameworks. For the question **Which \*Framework\* would you like to use for the client?**, you can see that there is only Vue.js presented as an option since we provided the blueprint. Also, you will see a message, **info Using blueprint generator-jhipster-vuejs for client subgenerator**, noting that the blueprint is being used.
5. Once all the prompts have been completed, JHipster will generate the application and start installing dependencies before starting the webpack build.

Our selected options will look as follows:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)
? What is the base name of your application? jhvue
? What is your default Java package name? com.mycompany.store
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? MySQL
? Which *development* database would you like to use? H2 with disk-based persistence
? Do you want to use the Spring cache abstraction? Yes, with the Ehcache implementation (local cache, for a single node)
? Do you want to use Hibernate 2nd level cache? Yes
? Would you like to use Maven or Gradle for building the backend? Gradle
? Which other technologies would you like to use? (Press <space> to select, <a> to toggle all, <i> to invert selection)
info Using blueprint generator-jhipster-vuejs for client subgenerator
? Which *Framework* would you like to use for the client? Vue.js
? Would you like to use a Bootswatch theme (https://bootswatch.com/)?
Default JHipster
info Using blueprint generator-jhipster-vuejs for common subgenerator
? Would you like to enable internationalization support? Yes
? Please choose the native language of the application English
? Please choose additional languages to install Hindi
? Besides JUnit and Jest, which testing frameworks would you like to use?
Protractor
? Would you like to install other generators from the JHipster Marketplace?
```

No

```
info Using blueprint generator-jhipster-vuejs for languages
subgenerator
```

Installing languages: en, hi

That's it; we are done. Our first JHipster Vue.js application was created successfully. Now, let's start the application so that we can play around with it.

Since we chose Gradle to build our application, let's start our server by running `./gradlew` in a Terminal.

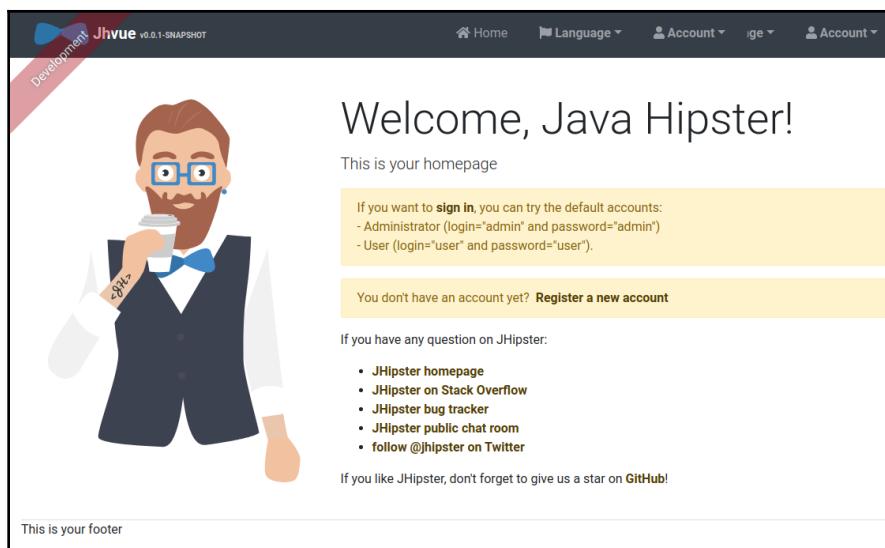
Gradle will download the necessary dependencies and start the Spring Boot application using the embedded Undertow container. Once the application starts successfully, we will see the following output in the console:

```
2018-03-04 16:37:48.096 INFO 4730 --- [ restartedMain]
com.mycompany.store.JhvueApp :

-----
Application 'jhvuedev' is running! Access URLs:
Local: http://localhost:8080
External: http://192.168.2.7:8080
Profile(s): [swagger, dev]
```

---

Visit the URL (`http://localhost:8080`) in your favorite browser to see the application in action:



You will see the preceding home screen, which is identical to the ones we saw for Angular and React.

Go ahead and log in using the default admin user and play around.

In the next section, we'll take a look at the technical stack.

## Technical stack and source code

Before we look at the generated code, let's talk about the technical stack. We looked at Vue.js in Chapter 2, *Getting Started with JHipster*, but let's recap.

Vue.js (<https://vuejs.org>) is a progressive JavaScript framework. It is open source and community-driven. Vue.js borrows concepts from both AngularJS (older version) and React. It has a similar syntax to AngularJS, but has the speed and performance of React. Like React, Vue.js is also a UI framework that can be combined with other libraries to build single-page applications. Vue.js can be written in JavaScript or TypeScript and can be used to write small web components or full-fledged SPA applications.

For example, the following is a simple Vue.js app:

```
<html>
  <head>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
  </head>
  <body>
    <div id="app">
      {{ message }}
    </div>
    <script>
      const app = new Vue({
        el: '#app',
        data: {
          message: 'Hello Vue Hipster!'
        }
      })
    </script>
  </body>
</html>
```

Generally, for Vue.js components, we will use `vue` as the file extension.

Similar to React, Vue also uses the virtual DOM concept to improve rendering efficiency. It also provides composable components and maintains its focus on rendering of the view layer, while other concepts such as routing and state management are left to other libraries. This is the major difference between React/Vue and Angular, where the latter is a full-fledged MVVM framework.

Due to this, when building Vue applications, we would always have to add a few more libraries for things such as state management and routing. Unlike React, Vue provides official libraries for state management and routing.

## Technical stack

The following are the technical stacks that are used by JHipster when Vue is chosen as the client-side framework:

- **Rendering:** Vue.js written using TypeScript and HTML
- **State management:** Vuex
- **Routing:** Vue router
- **HTTP:** Axios
- **Responsive design:** Bootstrap 4 + BootstrapVue
- **Linting:** TSLint
- **Unit testing:** Jest + vue-jest
- **Build:** Webpack

Let's look at some of the most important components of the stack.

## Using TypeScript

Like Angular and React, the generated client-side for Vue is also written in TypeScript. Along with static type checking, this makes development more efficient and less error-prone. The components that are written in TypeScript are used along with Vue template files.

## State management with Vuex

Vuex (<https://vuex.vuejs.org/guide/>) is the official state management solution for Vue.js and is Redux inspired. JHipster uses Vuex for state management with Vue.js. It is also possible to use other state management solutions such as Redux and MobX.

Vuex provides a global immutable store that can only be updated by committing mutations explicitly. The store is also reactive, which means the components that use the store data are updated when the store changes. A mutation is a method that changes data in the store. You need to call the store commit method with the required mutation.

Vue components automatically get access to the store when you connect a store to an app. Let's take a look at `src/main/webapp/app/core/home/home.component.ts`:

```
@Component
export default class Home extends Vue {
    @Inject('loginService')
    private loginService: () => LoginService;

    public openLogin(): void {
        this.loginService().openLogin(<any>this.$root);
    }

    public get authenticated(): boolean {
        return this.$store.getters.authenticated;
    }

    public get username(): string {
        return this.$store.getters.account ? this.$store.getters.account.login
            : '';
    }
}
```

As you can see, the component has `$store` as an instance variable. The stores that are used are defined under `src/main/webapp/app/shared/config/store`.

## Routing with Vue Router

Vue Router (<https://router.vuejs.org/>) is the official Vue.js router and we use it for client-side routing. The default setup with JHipster is to use browser history-based routing (HTML5 pushState). It supports component-based routing, along with an API for advanced routing setups. Routes are defined centrally using the Router object.

Let's take a look at `src/main/webapp/app/router/index.ts`:

```
Vue.use(Router);

export default new Router({
    mode: 'history',
    routes: [
        {

```

```
path: '/',
  name: 'Home',
  component: Home
},
{
  path: '/forbidden',
  name: 'Forbidden',
  component: Error,
  meta: { error403: true }
},
{
  path: '/not-found',
  name: 'NotFound',
  component: Error,
  meta: { error404: true }
},
{
  path: '/register',
  name: 'Register',
  component: Register
},
...
{
  path: '/account/password',
  name: 'ChangePassword',
  component: ChangePassword,
  meta: { authorities: ['ROLE_USER'] }
},
...
{
  path: '/admin/user-management',
  name: 'JhiUser',
  component: JhiUserManagementComponent,
  meta: { authorities: ['ROLE_ADMIN'] }
},
...
]);
});
```

All the routes that are used by the application are defined centrally here, unlike in React or Angular, where the definitions are split across modules.

## HTTP requests using Axios

Similar to our setup for React, Axios (<https://github.com/axios/axios>) is used to fetch data from the JHipster application's server-side REST endpoints from the Vue.js services. The resulting Promise is resolved by the service to provide data to the components.

The following shows a service using axios:

```
export default class UserManagementService {
    public get(userId: number): Promise<any> {
        return axios.get(`api/users/${userId}`);
    }

    public create(user): Promise<any> {
        return axios.post('api/users', user);
    }
    ...
}
```

The preceding service uses axios and returns a promise that can be resolved from the component.

## Bootstrap components using BootstrapVue

JHipster uses Bootstrap 4 as its UI framework. Since we are building a Vue application, it makes sense to use a native Vue binding instead of Bootstrap's jQuery-based components. BootstrapVue (<https://bootstrap-vue.js.org/>) provides pure Vue components and directives for Bootstrap 4.

We only register the components we require

in `src/main/webapp/app/shared/config/config-bootstrap-vue.ts`.

Let's take a look at `src/main/webapp/app/account/login-form/login-form.vue`:

```
<template>
    ...
    <b-alert show variant="danger" v-
        if="authenticationError" v-html="...">
        ...
    </b-alert>
</div>
<div class="col-md-8">
    <b-form role="form" v-on:submit.prevent="doLogin()">
        <b-form-group v-bind:label="..." label-
            for="username">
```

```
<b-form-input id="username" type="text" ...  
    v-model="login">  
</b-form-input>  
</b-form-group>  
...  
<b-form-checkbox id="rememberMe" name="rememberMe"  
    v-model="rememberMe" checked>  
    <span v-text="$t('login.form.rememberme')"  
        ">Remember me</span>  
</b-form-checkbox>  
<div>  
    <b-button type="submit" variant="primary"  
        v-text="$t('login.form.button')">Sign in  
</b-button>  
</div>  
</b-form>  
...  
</div>  
</div>  
</div>  
</template>  
...
```

The preceding template uses standard HTML for the markup, along with additional custom-elements (directives and components) that are defined using Vue.js.

## Unit testing setup

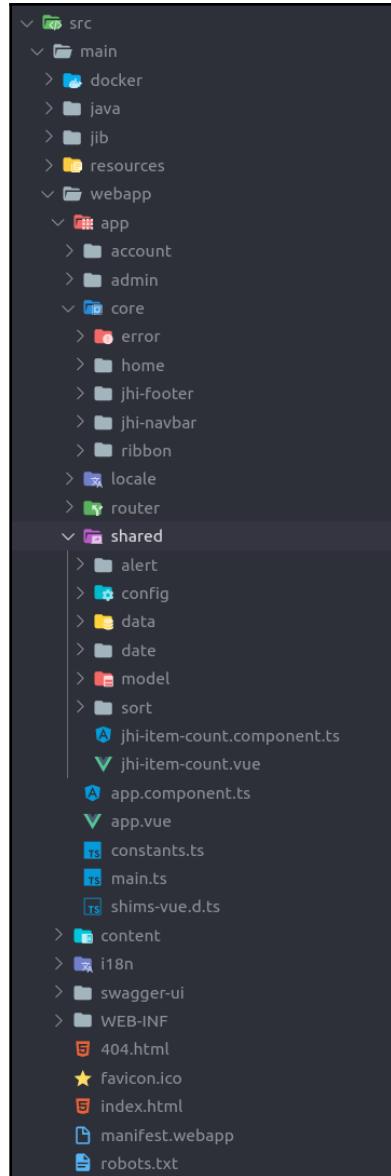
JHipster uses Jest to unit test the client-side components, similar to our React and Angular setup. Vue test utilities are used to mount and interact with Vue components.

The configuration for this can be found in `src/test/javascript/jest.conf.js`.

Let's run the generated unit tests. Run `npm test` in a Terminal.

## Generated source code

Let's take a look at the generated code. Since we looked at the server-side code in the previous chapters, we will only look at the client-side code here:



This structure is slightly different from what we saw for Angular and React. We are only concerned about the code inside `src/main/webapp/app` since everything else is exactly the same as what we saw for the Angular/React application.

Let's take a look at some of the important parts of the code:

- `main.ts`: This is the entry point of our application. This is where we bootstrap Vue to the `root` `div` and initialize the application, along with other dependencies such as Vuex store, Bootstrap, and `i18n`:

```
...
Vue.config.productionTip = false;
config.initVueApp(Vue);
config.initFontAwesome(Vue);
bootstrapVueConfig.initBootstrapVue(Vue);
Vue.use(Vue2Filters);
Vue.component('font-awesome-icon', FontAwesomeIcon);
Vue.component('jhi-item-count', JhiItemCountComponent);

const i18n = config.initI18N(Vue);
const store = config.initVueXStore(Vue);

const alertService = new AlertService(store);
const translationService = new TranslationService(store, i18n);
const loginService = new LoginService();
const accountService = new AccountService(store,
translationService, router);

...
/* tslint:disable */
new Vue({
  el: '#app',
  components: { App },
  template: '<App/>',
  router,
  provide: {
    loginService: () => loginService,
    ...
    accountService: () => accountService
  },
  i18n,
  store
});
```

- `app.component.ts`: This is our main application component:

```
...
@Component({
  components: {
    ribbon: Ribbon,
    'jhi-navbar': JhiNavbar,
    'login-form': LoginForm,
    'jhi-footer': JhiFooter
  }
})
export default class App extends Vue {}
```

- `app.vue`: This is our main application component template. We define the main application UI structure here:

```
<template>
  <div id="app">
    <ribbon></ribbon>
    <div id="app-header">
      <jhi-navbar></jhi-navbar>
    </div>
    <div class="container-fluid">
      <div class="card jh-card">
        <router-view></router-view>
      </div>
      <b-modal id="login-page" hide-footer lazy>
        <span slot="modal-title" id="login-title" v-‐
          text="$t('login.title')">Sign in</span>
        <login-form></login-form>
      </b-modal>

      <jhi-footer></jhi-footer>
    </div>
  </div>
</template>

<script lang="ts" src="./app.component.ts">
</script>
```

- `constants.ts`: Application constants.
- `router`: All of the application's routes are defined here and they are imported into `main.ts` from here.

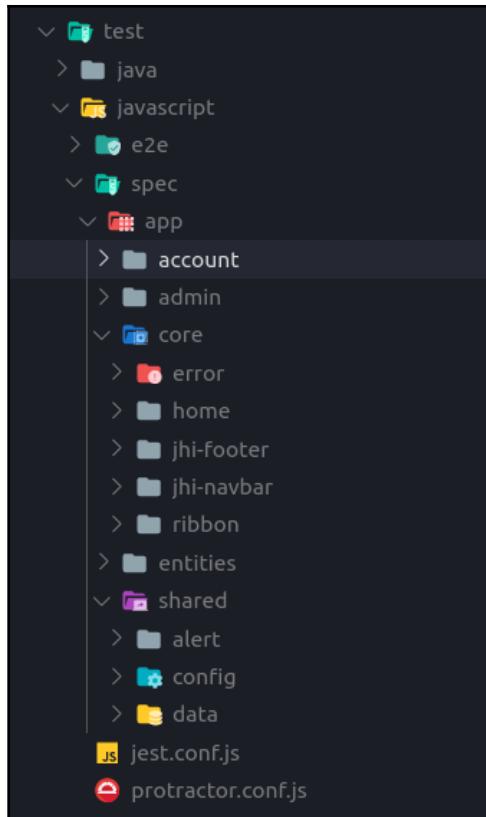
- account: The login and logout components, along with **Account** pages such as settings, and password reset, can be found here.
- admin: The admin screens, such as metric, health, and user management, are here.
- entities: The entity modules are present here.
- core: The core application UI modules, such as home, navigation bar, and error, can be found here.
- locale: Translation service used by the application.
- shared: Shared components and reducers:
  - alert: Alert mixins and services are defined here.
  - config: This is where framework-level configurations are done:
    - axios-interceptor.ts: HTTP interceptors are configured here. This is where the JWT tokens are set to requests and errors are handled.
    - config-bootstrap-vue.ts: Bootstrap components are declared here.
    - config.ts: Application configurations, such as fonts, store, i18n, and date formats, are executed here.
    - formatter.ts: Formatters used in the application.
    - store: Vuex stores are defined here:
      - account-store.ts: This is used for account-related actions. It defines the state and mutations that can be used:

```
export const accountStore: Module<any, any> = {
  state: {
    logon: false,
    userIdentity: null,
    authenticated: false,
    ribbonOnProfiles: '',
    activeProfiles: ''
  },
  getters: {
    logon: state => state.logon,
    account: state => state.userIdentity,
    authenticated: state =>
      state.authenticated,
  }
}
```

```
        activeProfiles: state =>
            state.activeProfiles,
        ribbonOnProfiles: state =>
            state.ribbonOnProfiles
    },
mutations: {
    authenticate(state) {
        state.logon = true;
    },
    authenticated(state, identity) {
        state.userIdentity = identity;
        state.authenticated = true;
        state.logon = false;
    },
    logout(state) {
        state.userIdentity = null;
        state.authenticated = false;
        state.logon = false;
    },
    setActiveProfiles(state, profile) {
        state.activeProfiles = profile;
    },
    setRibbonOnProfiles(state, ribbon) {
        state.ribbonOnProfiles = ribbon;
    }
}
};
```

- **data:** Data utility services are defined here.
- **date:** Date filters are defined here.
- **model:** TypeScript model for entities.

The folder structure of the unit test code is also quite similar:



Now, we'll generate an entity for this application.

## Generating an entity with VueJS client-side

Now, let's learn how to create an entity using the JHipster entity generator with a Vue.js client-side. We will create a simple employee entity with the name, age, and date of birth fields, just like we did for the React app earlier:

1. Open a Terminal, navigate to the folder of the Vue.js app, and run `jhipster entity employee`.
2. Create the fields one by one, select **Yes** for the question **Do you want to add a field to your entity?**, and fill in the name of the field with **name** for the next question, **What is the name of your field?**

3. Select **String** as the field type for the next question, **What is the type of your field?**
4. For the question **Which validation rules do you want to add?**, choose **Required** for the **name** field and proceed.
5. Continue this process for the **age** and **dob** fields. **age** is of the integer type, while **dob** is of the instant type.
6. When asked again, **Do you want to add a field to your entity?**, choose **No**.
7. For the next question, **Do you want to add a relationship to another entity?**, choose **yes**.
8. Provide **user** as the name of the other entity, and as the name of the relationship for the questions that follow.
9. For the next question, **What is the type of the relationship?**, we'll create a one-to-one relationship with the user.
10. Choose no for the next two questions and no again when asked to add another relationship.
11. For the questions that follow, select the default options and proceed.  
The `jhipster entity employee` command will produce the following console output:

```
Using JHipster version installed globally
Executing jhipster:entity employee
Options:

The entity employee is being created.

...
=====
Fields
name (String) required
age (Integer)
dob (Instant)

Relationships
user (User) one-to-one

? Do you want to use separate service class for your business logic? No, the REST controller should use the repository directly
? Is this entity read-only? No
? Do you want pagination on your entity? No

Everything is configured, generating the entity...
```

```
info Using blueprint generator-jhipster-vuejs for entity-
client
subgenerator
...
```

JHipster will generate the entity and run the webpack build. From the logs, you will see that the Vue.js blueprint is being used here.



As an exercise, why don't you try to use here the JDL model from the monolith application we built earlier to generate entities?

12. If your server is not running, start it in a Terminal by running `./gradlew`. If it is already running, then just compile the new code by running `./gradlew compileJava`; Spring DevTools will restart the app automatically.
13. Start BrowserSync in another Terminal by running `npm start` and check the employee entity we just created:

### Create or edit a Employee

Name

Age

Dob

User

Cancel Save

14. Create an entity to check that everything works fine:

Employees					<a href="#">+ Create a new Employee</a>
ID	Name	Age	Dob	User	
1	mobile Fish	77969	Dec 12, 2019, 1:48 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
2	Table Computer	5808	Dec 12, 2019, 5:09 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
3	seamless Metal deposit	5641	Dec 11, 2019, 4:53 PM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
4	parsing red Chips	18340	Dec 12, 2019, 1:21 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
5	Home Loan Account	706	Dec 12, 2019, 5:48 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
6	intangible	61185	Dec 12, 2019, 6:34 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
7	Gorgeous Frozen Pants	23277	Dec 12, 2019, 5:13 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
8	bluetooth	19967	Dec 12, 2019, 6:59 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
9	Planner Canyon	85994	Dec 12, 2019, 1:35 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
10	Table Small Frozen Sausages Planner	80839	Dec 12, 2019, 6:55 AM		<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>
11	Superman	30	Jan 1, 1985, 12:00 AM	admin	<a href="#">View</a> <a href="#">Edit</a> <a href="#">Delete</a>

Now, let's see what happens once we've created the entity:

- For the entity we created, JHipster generated/updated the following files:

```
info Using blueprint generator-jhipster-vuejs for entity-client subgenerator
create .jhipster/Employee.json
create src/main/resources/config/liquibase/changelog/20191212151024_added_entity_Employee.xml
create src/main/resources/config/liquibase/changelog/20191212151024_added_entity_constraints_Employee.xml
create src/main/resources/config/liquibase/fake-data/employee.csv
create src/main/java/com/mycompany/store/domain/Employee.java
create src/main/java/com/mycompany/store/repository/EmployeeRepository.java
create src/main/java/com/mycompany/store/web/rest/EmployeeResource.java
create src/test/java/com/mycompany/store/web/rest/EmployeeResourceIT.java
create src/test/java/com/mycompany/store/domain/EmployeeTest.java
conflict src/main/resources/config/liquibase/master.xml
? Overwrite src/main/resources/config/liquibase/master.xml? overwrite this and all others
  force src/main/resources/config/liquibase/master.xml
  force src/main/java/com/mycompany/store/config/CacheConfiguration.java
create src/main/webapp/app/entities/employee/employee-details.vue
create src/main/webapp/app/entities/employee/employee.vue
create src/main/webapp/app/entities/employee/employee-update.vue
  force src/main/webapp/app/core/jhi-navbar/jhi-navbar.vue
create src/main/webapp/app/entities/employee/employee-details.component.ts
create src/main/webapp/app/entities/employee/employee.component.ts
create src/main/webapp/app/entities/employee/employee.service.ts
create src/main/webapp/app/shared/model/employee.model.ts
create src/main/webapp/app/entities/employee/employee-update.component.ts
create src/test/javascript/spec/app/entities/employee/employee.component.spec.ts
create src/test/javascript/spec/app/entities/employee/employee-details.component.spec.ts
create src/test/javascript/spec/app/entities/employee/employee.service.spec.ts
create src/test/javascript/spec/app/entities/employee/employee-update.component.spec.ts
create src/test/javascript/e2e/entities/employee/employee.page-object.ts
create src/test/javascript/e2e/entities/employee/employee.spec.ts
create src/test/javascript/e2e/entities/employee/employee-details.page-object.ts
create src/test/javascript/e2e/entities/employee/employee-update.page-object.ts
  force src/main/webapp/app/router/index.ts
  force src/main/webapp/app/main.ts
create src/main/webapp/i18n/en/employee.json
  force src/main/webapp/i18n/en/global.json
create src/main/webapp/i18n/hi/employee.json
  force src/main/webapp/i18n/hi/global.json
```

- On the Vue.js client-side, we have the following files:

```
src/main/webapp/app/entities/employee/employee-details.vue
src/main/webapp/app/entities/employee/employee.vue
src/main/webapp/app/entities/employee/employee-update.vue
src/main/webapp/app/entities/employee/employee-details.component.ts
src/main/webapp/app/entities/employee/employee.component.ts
src/main/webapp/app/entities/employee/employee.service.ts
src/main/webapp/app/shared/model/employee.model.ts
src/main/webapp/app/entities/employee/employee-update.component.ts
src/test/javascript/spec/app/entities/employee/employee.component.spec.ts
src/test/javascript/spec/app/entities/employee/employee-details.component.spec.ts
src/test/javascript/e2e/entities/employee/employee.page-object.ts
src/test/javascript/e2e/entities/employee/employee.spec.ts
src/test/javascript/e2e/entities/employee/employee-details.page-object.ts
src/test/javascript/e2e/entities/employee/employee-update.page-object.ts
```

```
src/test/javascript/spec/app/entities/employee/employee.service.spec.ts  
src/test/javascript/spec/app/entities/employee/employee-update.component.spec.ts  
src/test/javascript/e2e/entities/employee/employee.page-object.ts  
src/test/javascript/e2e/entities/employee/employee.spec.ts  
src/test/javascript/e2e/entities/employee/employee-details.page-object.ts  
src/test/javascript/e2e/entities/employee/employee-update.page-object.ts
```

- The routes for the entity are updated in the `router/index.ts` file:

```
const Employee = () => import('../entities/employee/employee.vue');  
const EmployeeUpdate = () => import('../entities/employee/employee-update.vue');  
const EmployeeDetails = () =>  
    import('../entities/employee/employee-details.vue');
```

- The `employee.service.ts` file declares the services that are required for the entities to communicate with the backend server. The following code block shows this:

```
...  
const baseApiUrl = 'api/employees';  
  
export default class EmployeeService {  
    public find(id: number): Promise<IEmployee> {  
        return new Promise<IEmployee>(resolve => {  
            axios.get(`${baseApiUrl}/${id}`).then(function(res) {  
                resolve(res.data);  
            });  
        });  
    }  
  
    public retrieve(): Promise<any> {  
        return new Promise<any>(resolve => {  
            axios.get(baseApiUrl).then(function(res) {  
                resolve(res);  
            });  
        });  
    }  
  
    public delete(id: number): Promise<any> {  
        return new Promise<any>(resolve => {  
            axios.delete(`${baseApiUrl}/${id}`).then(function(res) {  
                resolve(res);  
            });  
        });  
    }  
}
```

```

        });
    }

    public create(entity: IEmployee): Promise<IEmployee> {
        return new Promise<IEmployee>(resolve => {
            axios.post(` ${baseApiUrl}`, entity).then(function(res) {
                resolve(res.data);
            });
        });
    }

    public update(entity: IEmployee): Promise<IEmployee> {
        return new Promise<IEmployee>(resolve => {
            axios.put(` ${baseApiUrl}`, entity).then(function(res) {
                resolve(res.data);
            });
        });
    }
}

```

- `employee.component.ts`, `employee-update.component.ts`, and `employee-detail.component.ts` declare the entity listing, entity update, and entity detail pages, respectively. Let's look at `employee.component.ts`. Here, we define the component and extend it with the required mixins. We also inject the required services, followed by the component methods that will be used by the template:

```

...
@Component
export default class Employee extends mixins(Vue2Filters.mixin,
AlertMixin) {
    @Inject('employeeService')
    private employeeService: () => EmployeeService;
    private removeId: number = null;
    public employees: IEmployee[] = [];
    public isFetching = false;

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

    public clear(): void {
        this.retrieveAllEmployees();
    }

    public retrieveAllEmployees(): void {
    ...
}

```

```
}

public prepareRemove(instance: IEmployee): void {
    this.removeId = instance.id;
}

public removeEmployee(): void {
    ...
}

public closeDialog(): void {
    (<any>this.$refs.removeEntity).hide();
}
}
```

The other components also follow a similar approach. Code wise, the Vue.js code is more similar to our Angular implementation.

## Summary

In this chapter, we learned about the general concepts of Vue.js, Vuex, and some other libraries on the Vue.js ecosystem. We also learned how to create a Vue.js app using JHipster and generated an entity for it. We saw how we can make use of TypeScript with Vue.js and also walked through the generated code. Finally, we ran and tested our created application.

In the next chapter, we will conclude this book with some best practices from the JHipster community and look at the next steps you should take in order to make use of what you've learned so far.

# 15

## Best Practices with JHipster

In the previous chapters of this book, we learned about JHipster and the various tools and technologies it supports in detail. These are the things we have learned so far:

- We learned to develop monolithic and microservice applications. We also learned about differences in the architecture and the reasons to choose one over the other.
- We created entities using JDL, and we customized the generated application for our business needs.
- We created a CI/CD setup using Jenkins.
- We deployed the monolith application to the Heroku cloud.
- We deployed the microservice architecture to the Google Cloud using Kubernetes and Docker.
- We learned about Spring Framework, Spring Boot, Angular, React, Vue.js, Docker, Kubernetes, and much more.

In this chapter, we will see what steps you need to take next to use what you have learned from this book, and we will also talk about some of the best practices, tips, tricks, and suggestions from the JHipster community. As core contributors to JHipster, we will also provide some insights and lessons in this chapter. The following are some of the topics that we will touch upon:

- The next steps to take
- The best practices to keep in mind
- Using JHipster modules

## The next steps to take

JHipster supports a lot of technologies, and learning about all of them would require an insane amount of time and effort; it cannot be done in a single book. Each technology would require a book of its own to learn and master it. If you are already familiar with the core concepts of web development, you will have a fairly good idea of how a JHipster application works by now. We hope this book gave you a good introduction to the technologies and JHipster itself. But this in itself isn't sufficient; you will have to keep learning more to become a master. The following are some of the things you can do to further hone your skills in web development using JHipster. But before that, we would recommend that you learn more about Spring Framework and the Angular (or React or Vue.js) ecosystem to complement what you have learned in this book.

## Adding a shopping cart for the application

In Chapter 5, *Customization and Further Development*, we saw how the generated application can be customized to make it look and behave like an e-commerce website. As we mentioned there, it is not enough to make the application truly usable. The following are some of the features that you can try to implement to make the application more feature-complete:

- Add a simple shopping cart feature on the client-side:
  1. Create a `ProductOrder` object to hold the `OrderItems`. The `ProductOrder` is related to the customer, so tag it to the customer using the details of the currently logged-in user.
  2. Add an **Add to cart** button to the product items in the list. On clicking the button, create a new `OrderItem` for the product and add the `OrderItem` to the `OrderItems` array of the `ProductOrder`. If the same product is clicked more than once, increase the quantity attribute of the existing `OrderItem`. Add a shopping cart dialog to list down all the `OrderItems` added to the `ProductOrder`. It can use a similar listing UI to the products or a simple table to show the product, total price, and quantity.
  3. Add a **View cart** button to the product list page to view the shopping cart dialog.

- Add an **Order Now** feature:
  1. Add an **Order Now** button to the product list page.
  2. On clicking the button, send the `ProductOrder` entity to the REST API to create a new `ProductOrder`. Use the `product-order.service.ts` for this.
  3. At the backend, modify the `save` method of `ProductOrderService.java` to create an invoice and shipment for the `ProductOrder` and save them all.
  4. Let's assume that we accept cash on delivery, so let's skip integration with a payment gateway for now.
- Send an order confirmation to the customer:
  1. JHipster comes with mail configuration and templates out of the box. You can configure your own SMTP server details in `src/main/resources/config/application-* .yml`. Refer to [http://www.jhipster.tech/tips/011\\_tip\\_configuring\\_email\\_in\\_jhipster.html](http://www.jhipster.tech/tips/011_tip_configuring_email_in_jhipster.html) for instructions on how to configure popular SMTP services.
  2. Create a new email template in `src/main/resources/templates/mail` for order confirmation. Provide the details of products, total price, and quantity in the email.
  3. Use the provided `sendEmailFromTemplate` method in `MailService.java` to send the email when an invoice is successfully created.
- Create a customer profile when registering a new user:
  1. Add fields to the registration page and create a customer entity for every user from the details automatically.

Try to apply the changes to the microservice application as well.

## Improving end-to-end tests

In Chapter 6, *Testing and Continuous Integration*, we saw that some of the e2e tests were commented out because of the difficulty in generating tests for an entity with a required relationship. Try to fix the tests with the following approach:

1. Add a method to delete entities after creation, similar to what we saw in Chapter 6, *Testing and Continuous Integration*, for the customer entity spec.

2. Uncomment the commented-out e2e tests in the files under `src/test/javascript/e2e/entities`.
3. Navigate the protractor to the related entity page and create a new item. If the related entity has required relationships, then follow the same approach and nest them until all the required entities are in place. This can be done in a `beforeAll` method of the test as well.
4. Now go back to the entity under test and see whether the test works fine.
5. Once the test is complete, delete the created entities in the `afterAll` method of the test.
6. Explore whether you can automate the creation of an entity item on the page object of the entity and use it when needed.

## Improving the CI/CD pipeline

In Chapter 6, *Testing and Continuous Integration*, when we created the `Jenkinsfile` using the ci-cd sub-generator, we commented out the deployment stage. Re-enable it and check whether the application is deployed to Heroku when you make new commits. In particular, you should do the following:

- See if you can add e2e tests to the pipeline.
- If your application is on GitHub, try to add Travis to the project using the ci-cd sub-generator.

## Create an e-commerce application with React or Vue.js

In the previous two chapters, we created a React- and Vue.js-based application. We used the JDL we designed for the monolith application to recreate the same with React and/or Vue.js on the client-side.

## Building a JHipster module

JHipster has two mechanisms to extend its features:

- A modules system, which lets users build their own Yeoman generators (<http://www.jhipster.tech/modules/creating-a-module/>) to complement JHipster
- A blueprint mechanism to customize the required parts of the code generated by JHipster



The difference between a **module** and a **blueprint** is that a blueprint lets you override certain parts of the generated application while JHipster scaffolds the remaining parts. For example, a blueprint can override the client-side code alone, while the server side is generated by JHipster. A module, on the other hand, can only change what is generated by JHipster, and so is more suitable for adding complementary features on top of the ones created by JHipster.

Try to build a module to add a simple, contact us page to your application.



You can use the JHipster module generator (<https://github.com/jhipster/generator-jhipster-module>) to scaffold a new module.

Let's see what the best practices are.

## The best practices to keep in mind

Over the years, the JHipster community has identified and adopted a lot of best practices from the technologies and tools it supports and from the general technical community. While JHipster has tried to follow these best practices in the code it creates, the following are some best practices, tips, and tricks that you as a user should follow.

## Choosing a client-side framework

When using JHipster, you have the option to choose between using Angular, React, and Vue.js as the client-side framework. Do not choose something just for its hype—choose it based on your requirement, team composition, and familiarity:

- If you come from a heavy Java/Spring background, then Angular or Vue.js will be much easier to follow and work with
- If your application requires heavy state management and shared state, then React or Vue.js would be a more natural fit
- If you are planning to build a native mobile client for your application, then React is a good choice for this, with React Native allowing you to reuse a lot of code between your web and mobile application
- If your application depends heavily on HTML pages produced by a design team or a third party, then Angular or Vue.js will be much easier to integrate than React
- If you are familiar with AngularJS, then you might feel at home with Vue.js

If you need a lot of widgets that are not part of standard Bootstrap, then use an existing widget library, such as PrimeNG or VMware Clarity, rather than assembling widgets from different origins; however, if you only need a few more widgets on top of Bootstrap, then stick to Bootstrap and use a Bootstrap-compatible widget for Angular, Vue, or React.

Regardless of what you choose, follow the guidelines and best practices from that project's community.

## Choosing a database option

JHipster provides support for many kinds of database, ranging from SQL to NoSQL. The following are some considerations when choosing a **database (DB)**:

- For most cases, an SQL DB would be more than sufficient, so if you do not see any reason to go with other NoSQL solutions, stick to SQL and choose from MySQL, Postgres, Oracle, MariaDB, and MS SQL:
  - If you are working in an enterprise with Oracle or MS SQL subscriptions, then it would make sense to choose them as you would benefit from the support and enterprise features provided; otherwise, you should go with an open source solution.

- If you need to store and query a lot of JSON data, then Postgres offers the best JSON support with full-text search capabilities.
- For most simple use cases, MySQL or MariaDB will suffice.
- Always choose a second-level Hibernate cache when working with a SQL DB.
- When choosing a development database for SQL, you should do the following:
  - Choose an H2 file DB if you want a simple development setup with persistent data.
  - Choose the same DB as the production DB if you want faster restarts and your persistent data doesn't need to be wiped every now and then. If you are using the provided Docker images, then wiping data will not be an issue.
  - Choose an H2 in-memory DB if you do not want any persistent data during development and would like a clean state on each restart.
- If your use case requires a lot of heavy data reads/writes, and if the data is not very relational, then Cassandra would be a perfect fit, as it is distributed and can work under extremely heavy loads.
- For a normal, nonrelational data structure, MongoDB may be sufficient. You could also use Postgres as a NoSQL JSON store if needed.
- If you need enterprise support for NoSQL, then CouchBase is a good option.
- Use Elasticsearch along with the primary DB for full-text search. If you only need simple filtering, use the JPA filtering option provided. Refer to <http://www.jhipster.tech/entities-filtering/> for more information.
- Do not use Elasticsearch as a primary database if it is not built for that purpose.

## Architecture considerations

We have already discussed choosing a microservice or monolithic architecture in Chapter 1, *Introduction to Modern Web Application Development*. Here are some more points when it comes to architecture:

- Don't use a microservice architecture if you're a small team. Microservices are about scaling teams more than anything. It's often easier to break up your monolith than start with microservices.
- Use asynchronous messaging in your monolith if you think you may need to refactor to microservices in the future. JHipster provides support for Apache Kafka, which is a good solution for asynchronous messaging.



Asynchronous messaging is the best way of building stateless systems. It is important in a microservice architecture, as you might often want communications to be stateless and nonblocking. Some of the popular solutions for this are Apache Kafka (<http://kafka.apache.org/>), RabbitMQ (<https://www.rabbitmq.com/>), and gRPC (<https://grpc.io>). ReactiveX (<http://reactivex.io/>) and Spring Reactor (<http://projectreactor.io/>) are popular abstractions for working with asynchronous systems. Asynchronous messaging also makes the systems loosely coupled.

- If you intend to expose an API to a third party, use *API-first* development. We now have a good workflow to do this with the OpenAPI generator (<https://github.com/OpenAPITools/openapi-generator>). Refer to <http://www.jhipster.tech/doing-api-first-development/> for more information.
- When setting up communication between microservices with REST, don't put interface code in a shared package; it would tightly couple APIs to their clients, thereby creating a distributed monolith. It is better to have duplicated code than to be tightly coupled.
- With JHipster, it is possible to split the client and server. Refer to <http://www.jhipster.tech/separating-front-end-and-api/> for more information. However, think twice before separating them, as this will require you to open up CORS, which makes the security more vulnerable, and such architecture brings its own issues. So do this only if you have a good reason to.
- Use DTOs at the service layer so that you can aggregate entities and define a better API without exposing entities to the client. You will have to enable the service layer for your entities to use this with JHipster.
- Learn the technology stack of your application before you start development.

- Make yourself familiar with the provided toolbelt, such as build tools (Maven/Gradle/Webpack), BrowserSync, and so on.
- Follow the 12 factors of application development (<https://12factor.net/>).

## Security considerations

Security is one of the most important aspects of any application, and you should consider the following when choosing a security mechanism:

- For most use cases, JWT authentication will be sufficient, so stick to that if you are not sure
- If you want single-sign-on capabilities in your application, use OAuth 2.0/OIDC rather than trying to make JWT or session authentication work as an SSO solution
- If you already have Keycloak or Okta set up in your company, choose OAuth 2.0/OIDC and connect to it
- Choose session-based authentication only if you want a stateful authentication
- Do not open up CORS unless you have to
- Use Spring Security to add authorization logic to your API endpoints and services
- Remove all secrets from the `application-prod.yml` file and use placeholders to inject values from the command line or environment variables. Never put any secrets or passwords in code or config files
- Change the generated JWT secrets for production

Refer to <https://www.jhipster.tech/security/> for more about security in JHipster.

## Deployment and maintenance

There are a lot of good practices in the field of deployment and maintenance; some of the most important ones are as follows:

- Docker is a must-have for the integration testing of microservices, but going into production with Docker is not easy, so you should use an orchestration tool, such as Kubernetes or OpenShift, for that.
- Try to use Docker images even if you are building a monolith.

- Run a production build immediately after the application is generated and deploy to production immediately while your app is still very simple. This will help ease any deployment issues, as you will be sure that the app works fine out of the box.
- The production build is quite different from the development build when it comes to the client-side, as the resources are minified and optimized. When adding any new frontend code or libraries, always verify the production build as well to ensure that it works fine.
- Run prod builds often, and when you do, always run them in prod mode in CI/CD.
- Try to keep your client-side JS bundles small, as this will affect the performance. Similarly, only add a dependency if you must.
- Always run end-to-end protractor tests with the prod profile.
- Embrace the embedded servlet engine and forget about deploying to a JEE server such as WebLogic, WebSphere, JBoss, and so on. The artifacts produced are executable and have an embedded Undertow server.



Did you know that Java EE was renamed to Jakarta EE? Refer to <https://www.infoq.com/news/2018/03/java-ee-becomes-jakarta-ee> for more information.

- Upgrade the application that often uses the JHipster upgrade sub-generator. This will ensure that the tools and technologies that you use are up to date and secure. Incremental upgrades will reduce merge conflicts and keep the process simple.
- Add unit, integration, and e2e tests as much as possible. There is no such thing as too many tests.
- Set up CI/CD for the application. An application with a decent CI/CD setup will save you countless hours spent on regressions and will improve your delivery speed.
- Run CI builds when you commit to the master. Enable CI builds for PRs and merge PRs only when they pass. Following this, as a rule, will tremendously increase your code quality.

## General best practices

In general, the following are some best practices with respect to JHipster that you should consider:

- If you start creating entities using the entity sub-generator, then use `export-jdl` and switch to JDL once you have more than a handful of entities.
- Generate your application without any JHipster modules first and add the required modules only when the need arises.
- Evaluate a module carefully before adding it. Make sure that it supports the stack you have chosen.
- Follow each underlying technology's *best practices*—Angular best practices, Spring best practices, and so on. Change something only if there is a good reason to do so.
- Use the provided library versions on the client-side and server side. It's hard work to get them all working together, so stick to them. Update them only when JHipster updates them or if you really need to fix a bug or a security issue.
- Follow the workflows provided by JHipster. They are here to help you. There is usually a very good reason to use them in the recommended way. Read the JHipster documentation before looking for help outside.
- You have a great working environment out of the box; don't break it:
  - Frontend and backend updates are automatic and fast, using live reload. Make use of them.
  - Production deployment is easy using the provided sub-generators.
- Use the provided sub-generators for the cloud platform you are deploying to.
- Git is your friend. Commit each time you add a module or an entity, or when using a sub-generator. Every mistake (including those made in the database) should be easy to roll back with Git.

## Using JHipster modules

JHipster modules and blueprints are a great way to add more features and functionality to your generated code. There are many modules and blueprints available to choose from in the JHipster marketplace (<http://www.jhipster.tech/modules/marketplace>), and you can also build your own modules to suit your needs. Some of the modules worth noting are as follows:

- **Vue.js:** This module provides Vue.js support for JHipster applications. It creates a full-fledged client-side app for JHipster using Vue.js.

- **Ignite JHipster:** This provides a React Native boilerplate for JHipster apps, an ideal way to kickstart your React Native application using JHipster as the backend.
- **Entity Audit:** This module enables entity audits. It uses Hibernate audit hooks to create a custom audit for entity CRUD operations. It also provides JaVers as the auditing mechanism instead of the custom Hibernate auditing. It also provides a nice UI to view the audits in an Angular application. It will enable auditing for new entities as well as for existing entities.
- **Ionic:** This provides an Ionic client for JHipster apps. It is an ideal solution if you want to create mobile applications with a JHipster backend and an Angular frontend, made with the Ionic framework.
- **Swagger CLI:** This module provides support for generating Swagger clients for a JHipster application.
- **gRPC:** This module generates gRPC reactive endpoints for a JHipster application. It supports entities as well, and is an ideal choice if you want a nonblocking reactive API for your JHipster application.
- **Kotlin:** This blueprint uses Kotlin instead of Java for the backend code. If you prefer Kotlin over Java, then this is for you.
- **PrimeNG:** This module adds support for PrimeNG components in a JHipster Angular application.
- **NodeJS:** This blueprint makes it possible to use TypeScript as the backend running on NodeJS instead of Java.

There are many more modules and blueprints out there.

To use a module, follow these steps:

1. To use a JHipster module, first install it using `npm i -g generator-<module-name>` or `yarn add global generator-<module-name>`.
2. Once installed, go into the JHipster application directory and execute `yo <module-name>` to initiate the module and follow the prompts.

To use a blueprint, follow these steps:

1. To use a JHipster blueprint, first install it using `npm i -g generator-<blueprint-name>` or `yarn add global generator-<blueprint-name>`.
2. Once installed, go into a directory and execute `jhipster --blueprints <blueprint-name>` to initiate JHipster with the blueprint hooked to it and follow the prompts.

# Contributing to JHipster

One of the best ways to learn JHipster and the technologies it supports is by contributing to JHipster directly. Refer to the contribution guide (<https://github.com/jhipster/generator-jhipster/blob/master/CONTRIBUTING.md>) for details about setting up JHipster for development.

You can contribute to the project in many ways, such as the following:

- If you find a bug, enter an issue in the GitHub project (<https://github.com/jhipster/generator-jhipster>), follow the guidelines in the issue template, run `jhipster info`, and provide the steps to reproduce it. You can also try to fix the issue yourself and submit a PR if you're successful.
- Work on open issues and feature requests. This way, you will learn the internals of JHipster and the technologies used along the way.
- Answer JHipster-related questions on Stack Overflow (<https://stackoverflow.com/questions/tagged/jhipster>).

# Summary

Our journey together through JHipster and full stack development has come to an end. In this chapter, we learned about the many best practices identified by the JHipster community. Try to complete the assignments in the section called *The next steps to take*, as it will help you to apply what you have learned and will help you understand the concepts better.

*"Nothing helps you learn as much as trying and failing."*

We hope you have had a fabulous learning experience, and that what you have learned about JHipster from this book will help you with your next project.

Follow @jhipster on Twitter (<https://twitter.com/jhipster>) so that you can see when new releases come out and security vulnerabilities are revealed. Follow us on Twitter at @deepu105 (<https://twitter.com/deepu105?lang=en>) and @sendilkumarn (<https://twitter.com/sendilkumarn?lang=en>) as well, if you like.

If you have questions or issues regarding JHipster, then post your questions to Stack Overflow (<https://stackoverflow.com/questions/tagged/jhipster>) and add the `jhipster` tag. The team will be notified and will be happy to help!

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

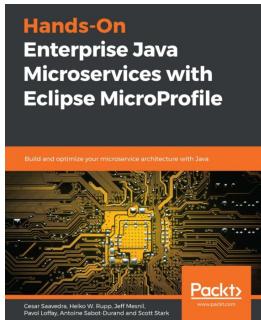


## **Hands-On Microservices with Spring Boot and Spring Cloud**

Magnus Larsson

ISBN: 978-1-78961-347-6

- Build reactive microservices using Spring Boot
- Develop resilient and scalable microservices using Spring Cloud
- Use OAuth 2.0/OIDC and Spring Security to protect public APIs
- Implement Docker to bridge the gap between development, testing, and production
- Deploy and manage microservices using Kubernetes
- Apply Istio for improved security, observability, and traffic management



## **Hands-On Enterprise Java Microservices with Eclipse MicroProfile**

Jeff Mesnil, Cesar Saavedra, Et al

ISBN: 978-1-83864-310-2

- Understand why microservices are important in the digital economy
- Analyze how MicroProfile addresses the need for enterprise Java microservices
- Test and secure your applications with Eclipse MicroProfile
- Get to grips with various MicroProfile capabilities such as OpenAPI and Typesafe REST Client
- Explore reactive programming with MicroProfile Stream and Messaging candidate APIs
- Discover and implement coding best practices using MicroProfile

## **Leave a review - let other readers know what you think**

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

# Index

## A

- account modules
  - about 81
  - password 82
  - registration 82
  - settings 81
- admin modules
  - about 83
  - API 88
  - audits 86
  - configuration 86
  - health 85
  - logs 87
  - metrics 84
  - user management 84
- Advanced Message Queuing Protocol (AMQP)
  - reference link 38
- Ahead-of-Time (AOT) compilation 28
- Amazon Web Services (AWS)
  - about 187
  - reference link 187
- Angular currency pipe
  - reference link 128
- Angular frontend
  - customizing, for entity 122, 124, 127
- Angular
  - about 28
  - URL 28
- application architecture 216, 217
- application generation
  - about 50
  - code, generating with JHipster 50
  - workspace, preparing 50
- application modules
  - about 78
  - account modules 81
- home and login modules 78, 79, 80
- application
  - client-side source code 74, 75, 76
  - code 61
  - deploying, to Docker 277
  - deploying, to Google Cloud 318, 319, 320, 321, 322
  - file structure 62, 63, 64
  - generating 316, 317
  - generating, with import-jdl 228, 229
  - modeling, with JDL 218
  - packaging, for local deployment 180
  - scaling up, with Docker 288
  - server-side source code 64
  - starting 77, 78
- architecture patterns
  - reference link 20
- Aspect-Oriented Programming (AOP)
  - URL 31
- async/await feature 338
- authorization, setting up with Spring Security
  - access, limiting to create/edit/delete entities 146
  - access, limiting to entities 145
  - access, limiting to other users data 147, 148, 150
- authorization
  - setting up, with Spring Security 145
- Axios
  - reference link 333, 355
  - used, for HTTP requests 355
- Azure App Service
  - about 188
  - JHipster application, deploying 188
  - reference link 188
- Azure Pipelines
  - about 160
  - URL 160

Azure Spring Cloud  
about 188  
reference link 188

## B

backend stack 9  
behavior-driven development (BDD) tests 38, 153  
best practices, JHipster  
about 374  
architecture considerations 377  
client-side framework, selecting 375  
database option, selecting 375, 376  
deployment 378  
general best practices 380  
maintenance 379  
security considerations 378  
Bill Of Materials (BOM) 34  
Bootstrap application  
customizing 141, 142  
Bootstrap button group  
reference link 130  
Bootstrap list group  
reference link 124  
Bootstrap row  
reference link 127  
Bootstrap  
about 27  
URL 27  
BootstrapVue  
reference link 355  
used, for providing Bootstrap components 355  
Bootswatch  
URL 27, 57, 141  
Boxfuse  
reference link 187  
BrowserSync  
about 29, 121  
URL 29, 121  
build tools  
BrowserSync 29  
Webpack 29

## C

caching 35  
Cascading Style Sheets (CSS) 26

Cassandra  
about 41  
URL 41  
cherry-picking advanced options  
reference link 138  
CI/CD pipeline  
improving 373  
CI/CD tools  
about 159  
Azure Pipelines 160  
GitHub Actions 161  
GitLab CI 161  
Jenkins 160  
Travis CI 160  
client side, entity generation  
Angular components 113, 114  
Angular module 115  
Angular Route 114  
Angular services 113  
TypeScript model class 112  
client-side discovery pattern 199  
client-side end-to-end tests 153  
client-side source code 74, 75, 76  
client-side stack 9  
client-side technologies, JHipster  
about 25  
Bootstrap 27  
build tools 29  
CSS3 26  
flexbox 26  
HTML5 26  
internationalization (i18n) 30  
MVVM framework 27  
SASS 26  
testing tools 29  
client-side tests 89, 90  
client-side unit tests 153  
Cloud Foundry  
about 187  
reference link 187  
code generation, with JHipster  
client-side options 56, 57  
internationalization options 58  
modules 59, 60, 61  
server-side options 52, 53, 54, 55

testing options 59  
components, stacks  
  Axios 355  
  BootstrapVue 355  
  TypeScript, using 352  
  Vue Router 353  
  Vuex 352  
Consul  
  about 205  
  advantages 205  
  health discovery 205  
  Key/Value store 206  
  multiple data centers 206  
  service discovery 205  
  URL 205  
continuous integration 158, 159  
Couchbase  
  about 41  
  URL 41  
Cross-Origin Resource Sharing (CORS) 69  
Cross-Site Request Forgery (CSRF) 235  
CSS3  
  reference link 26  
Cucumber  
  about 38  
  URL 38  
Curator 281  
custom resource definitions (CRD) 315

## D

Data Transfer Object (DTO)  
  about 73, 92, 95  
  reference link 95  
database options  
  about 38  
  NoSQL databases 40  
  SQL databases 39  
deployed application  
  features 283, 284  
deployment options, JHipster  
  Amazon Web Services (AWS) 187  
  Azure App Service 188  
  Azure Spring Cloud 187  
  Cloud Foundry 187  
  Google App Engine (GAE) 187

  Heroku 186  
DevOps 9  
distributed tracing system libraries 202  
Docker Compose  
  about 176, 267  
  commands 176  
Docker containers 174  
Docker Hub  
  about 175  
  reference link 175  
docker-compose files  
  generating, for microservices 279, 280, 281, 282  
Docker  
  about 44, 173  
  application, building 277  
  application, deploying to 277  
  application, scaling up 288  
  commands 176  
  files 273, 274, 276  
  production database, starting with 177  
  used, for building application 180  
  used, for deploying application 180  
Dockerfiles  
  about 174  
  reference link 175  
Document Object Model (DOM) 125  
domain-specific language (DSL) 91  
DSL grammar  
  for JDL 92

**E**

e-commerce application  
  creating, with React 373  
  creating, with Vue.js 373  
Ehcache  
  about 35  
  URL 35  
Elastic Stack  
  reference link 25  
Elasticsearch  
  about 41, 208  
  URL 24, 41  
ELK Stack 25  
end-to-end tests

improving 372

**Entity Audit** 381

**entity generation**

- client-side 112
- server-side source code 107

**entity modeling**

- with JDL 93, 94

**entity**

- Angular frontend, customizing 122, 124, 127, 129
- defining 98, 99, 100
- editing, with JHipster entity sub-generator 133, 136, 138, 139
- generating, with JHipster 104, 105
- generating, with React client side 340, 342, 344, 346
- generating, with Vue.js client-side 362, 364, 367, 369
- options 102, 103

**Envoy proxies**

- reference link 310

**Enzyme**

- reference link 334

**ES6+ features**

- URL 28

**ES6**

- URL 28

**Eureka client** 203

**Eureka server** 203

**executable archive**

- building 182
- deploying 182

## F

**fault tolerance libraries** 202

**filter option**

- reference 96

**filtering functionality** 131, 132, 133

**flexbox**

- reference link 26

**Flux**

- reference link 330

**frontend stack** 9

**full-stack developer** 9

## G

**gateway application**

- about 219, 230, 231, 232
- entities 234
- JDL specification 220, 221

**Gatling**

- about 38
- URL 38

**generated Docker Compose files** 272

**generated files**

- exploring 298, 299, 300

**generated pages** 115, 116, 117, 118

**generated source code** 357, 359, 361

**generated tests**

- running 88, 118

**Gherkin** 38

**Git** 43

**Git commands**

- reference link 50

**Git flow**

- reference link 135

**GitHub Actions**

- about 161
- URL 161

**GitHub flow**

- reference link 135

**GitKraken**

- URL 61

**GitLab CI**

- about 161
- URL 161

**Google App Engine (GAE)**

- about 187
- reference link 187

**Google Cloud Platform (GCP)** 314

**Google Cloud**

- application, deploying to 318, 319, 321, 322

**Gradle**

- about 34
- URL 34

**Grafana**

- reference link 311

**gRPC** 381

# H

- H2
    - about 39
    - URL 39
  - Hazelcast
    - about 36
    - URL 36
  - health check service
    - about 200
    - pull configuration 200
    - push configuration 200
  - Helm
    - reference link 311
  - Heroku CLI tool
    - installation link 189
  - Heroku Cloud
    - production deployment 189, 190, 192
  - Heroku
    - about 186
    - reference link 186, 189
  - Hibernate
    - about 34
    - URL 34
  - high availability system 200
  - HMAC SHA256
    - reference link 211
  - home and login modules 78, 79
  - HTML5 Boilerplate
    - about 26
    - reference link 26
  - HTML5
    - about 26
    - reference link 26
  - Hype Driven Development
    - reference link 19
  - HyperText Markup Language (HTML) 26
  - Hystrix
    - about 207
    - functionalities 207
    - reference link 207
- 
- I
  - i18n language
    - adding 143, 144
- Ignite JHipster 381
  - Infinispan
    - about 36
    - URL 36
  - internationalization (i18n) 30, 58
  - Inversion of Control (IoC)
    - URL 31
  - invoice microservice configuration 234, 235, 236, 237, 238
  - Ionic 381
  - Istio service mesh
    - microservice 311, 313
    - using 309
  - Istio
    - about 309
    - control plane 310
    - data plane 310
    - deploying, to Kubernetes cluster 314, 315, 316
    - functionalities 310
    - planes 310
    - reference link 309

# J

- Jaeger
  - reference link 311
- Java 42
- Java Message Service (JMS)
  - URL 38
- Java Persistence API (JPA) 34
- Java source 66, 67, 69, 71, 72
- Java-based JSON Web Tokens (JWT) 212
- JDL specification
  - for microservice invoice application 222
  - for microservice notification application 223
- JDL-Studio
  - about 96
  - reference link 96, 97, 220
- Jenkins pipeline
  - creating, with JHipster 162, 163
- Jenkins server
  - Jenkinsfile, setting up 167, 169, 170, 171
- Jenkins
  - about 160
  - setting up 161
  - URL 160

**Jenkinsfile**  
about 164, 165, 167  
setting up, in Jenkins server 167, 169, 170, 171  
stages 164, 165, 167

**Jest**  
about 30  
URL 30, 334

**JHipster Console**  
about 208  
demo 284, 285, 286, 287  
Elasticsearch 208  
Kibana 209  
Logstash 208  
reference link 208, 288  
Zipkin 209

**JHipster Domain Language (JDL)**  
about 91  
application, modeling 218  
DSL grammar 92  
DTO 95  
microservice entities, modeling 223, 224, 226, 228  
microservice stack, generating 217  
pagination options 95, 96  
reference link 91, 217  
relationship management 94, 95  
service option 95, 96  
service options 96  
using, for entity modeling 93, 94

**JHipster entity sub-generator**  
used, for editing entity 133, 135, 137, 140

**JHipster gateway**  
about 206  
Hystrix 207  
Netflix Zuul 206

**JHipster Marketplace**  
reference link 59

**JHipster modules**  
building 374  
Entity Audit 381  
gRPC 381  
Ignite JHipster 381  
Ionic 381  
Kotlin 381  
NodeJS 381

**PrimeNG** 381  
**Swagger CLI** 381  
using 380  
**Vue.js** 380

**JHipster Registry**  
about 202, 217  
application listing page 250  
configuration page 253  
dashboard 249  
Docker mode 244  
Eureka server 241  
general info 250  
health 250  
health page 253  
instances registered 250  
loggers' screen 255  
logs page 254  
metrics page 251  
Netflix Eureka 203  
pages 248  
pre-packaged WAR file, using 241  
reference link 203  
setting up, locally 240  
Spring Cloud Config Server 204  
Spring Cloud Config server 241  
starting, from Docker mode 244, 245  
Swagger API endpoints 255, 258, 260  
system status 249

**JHipster UAA server** 51, 212

**JHipster VueJS blueprint**  
reference link 27

**JHipster, prerequisites**  
about 42  
Docker 44  
Git 43  
Java 42  
Node.js 43  
tools 42

**JHipster-UML**  
reference link 92

**JHipster**  
about 22  
adoption 24  
benefits 23  
best practices 374

- client-side technologies 25  
contributing to 382  
goal 24  
IDE configuration 45  
installing 41, 46, 47  
Kubernetes configuration files, generating 291  
reference link 140  
server-side technologies 30  
setting up 41  
supported deployment options 186  
supported options 202  
supported technologies 25  
system setup 45  
upgraded sub-generator, reference link 182  
used, for creating Jenkins pipeline 162, 163  
used, for creating React application 325, 326,  
  327, 328  
used, for entity generation 104, 105  
version, upgrading 182, 184, 186
- JRebel 121
- JSON Web Token (JWT)  
  about 33  
  compact 211  
  self-contained 211  
  URL 33  
  working 212
- JSX  
  reference link 328
- JUnit  
  about 38  
  URL 38
- Just-in-Time (JIT) compilation 75
- JWT authentication 211
- K**
- Kafka  
  about 37  
  URL 37
- Keycloak  
  URL 34
- Kiali  
  reference link 311
- Kibana  
  about 209  
  reference link 209
- Kotlin 381  
Kotlin DSL  
  URL 34
- kubectl  
  reference link 291
- Kubernetes cluster  
  Istio, deploying to 314, 315, 316
- Kubernetes configuration files  
  generating, with JHipster 291
- Kubernetes manifests  
  generating 291, 292, 293, 295, 297
- Kubernetes  
  about 269, 271, 290  
  application, deployment to Google Cloud 301,  
    303, 304, 306, 307, 308
- L**
- Linux containers  
  reference link 174
- Liquibase  
  about 35  
  URL 35
- live reload  
  BrowserSync, using 121  
  for development 120  
  setting up, for application 122  
  Spring Boot DevTools, using 120
- Logstash  
  about 208  
  reference link 208
- Lucene  
  URL 41
- M**
- man-in-the-middle attacks  
  reference link 201
- MapReduce 40
- Mapstruct  
  reference link 95
- MariaDB  
  about 39  
  URL 39
- materia 141
- Maven  
  about 34

URL 34

Memcached

- about 36
- URL 36

Micrometer

- about 37
- URL 37

microservice application 51

microservice applications, versus monoliths

- applications
- about 195
- efficiency 196
- scalability 195
- time constraint 196

microservice architecture

- about 12, 16
- dynamic routing 200
- example 16, 17
- failover 202
- fault tolerance 201
- fundamentals 197
- health check 200
- resiliency 200
- security 201
- selecting 20
- service discovery 198
- service registry 198

microservice deployment options

- about 267
- Docker Compose 267, 269
- Kubernetes 269, 271
- OpenShift 272

microservice entities

- modeling, in JDL 223, 225, 226, 228

microservice gateway 51

microservice invoice application

- about 222
- JDL specification 222

microservice notification application

- about 222
- JDL specification 223

microservice stack

- executing, locally 245
- gateway application pages 245, 247, 248
- generated pages 262, 263

generating, with JDL 217

invoice application, executing locally 260, 262

notification applications, executing locally 260, 262

microservice web application architecture

- advantages 18
- disadvantages 18

microservice

- with Istio service mesh 313

microservices

- docker-compose files, generating for 279, 280, 281, 282
- requisites 213
- with Istio service mesh 311

Minikube 271

mobile-first web development 27

MobX

- URL 28

modern full-stack web development 10, 11

MongoDB

- about 40
- URL 40

monolithic application 51

monolithic architecture 13

- about 12
- selecting 19

monolithic web application architecture

- advantages 15
- disadvantages 15

MS SQL

- about 40
- URL 40

MVVM framework 27

MySQL

- about 39
- URL 39

**N**

Netflix Eureka

- reference link 199, 203

Netflix OSS

- URL 24

Netflix Zuul

- about 206
- reference link 207

Node.js 43  
NodeJS 381  
NoSQL databases  
  about 40  
  Cassandra 41  
  Couchbase 41  
  Elasticsearch 41  
  MongoDB 40  
notification microservice configuration 238, 239

## O

OAuth2  
  about 33, 212  
  URL 33  
object-relational mapping (ORM) 34  
Okta  
  URL 34  
OpenAPI specification 36  
OpenID Connect  
  about 33  
  URL 33  
OpenShift 272  
Oracle  
  about 40  
  URL 40

## P

performance tests 153  
Pivotal Cloud Foundry (PCF) 187  
Pivotal Web Services (PWS) 187  
Platform as a Service (PaaS) 160, 188  
pod 270  
PostgreSQL  
  about 40  
  URL 40  
pre-packaged WAR file  
  download link 241  
  using 244  
PrimeNG 381  
production database  
  starting, with Docker 177, 178  
Project Object Model (POM) 34  
Prometheus  
  about 209  
  benefits 210

components 210  
reference link 311  
versus JHipster Console 210  
Protractor  
  about 30  
  URL 30  
proxy microservice pattern 216

**R**

React application  
  creating, with JHipster 325, 327, 328  
React client side  
  used, for generating entity 340, 342, 344, 346  
React Router  
  URL 28, 331  
React  
  about 28  
  e-commerce application, creating 373  
  URL 28  
Reactive microservice application 51  
Reactive monolithic application 51  
reactstrap  
  URL 27, 333  
Redis  
  about 36  
  URL 36  
Redux Promise Middleware  
  reference 331  
Redux Thunk  
  reference 331  
Redux  
  URL 28, 330  
relational database management systems (RDBMS) 39  
relationships  
  adding 101, 102  
REpresentational State Transfer (REST) 13

**S**

server-side discovery pattern 199  
server-side integration tests 153  
server-side source code, entity generation  
  domain class, for entity 107, 108, 109, 110  
  repository interface, for entity 110  
  resource class, for entity 111

service class, for entity 110  
server-side source code  
about 107  
Java source 66, 67, 69, 71, 72  
resources 73  
server-side stack 9  
server-side technologies, JHipster  
about 30  
build tools 34  
caching 35  
Hibernate 34  
Kafka 37  
Liquibase 35  
security 33  
Spring Framework 31  
Swagger 36  
testing frameworks 38  
Thymeleaf 37  
WebSocket 37  
server-side tests 89  
server-side unit tests 153  
service discovery  
about 198  
client-side discovery pattern 199  
server-side discovery pattern 199  
service registry 198  
session policies  
reference link 235  
session-based authentication 33  
shopping cart  
adding, for application 371, 372  
Simple Object Access Protocol (SOAP) 13  
single page applications (SPAs) 28  
sorting functionality 129, 130  
source code  
about 328  
generating 334, 336, 338, 340  
Sourcetree  
URL 61  
Spring Boot DevTools  
about 120  
reference link 120  
Spring Boot  
about 31  
reference link 31  
Spring Cache abstraction  
reference link 35  
Spring Cloud Config Server  
about 204  
reference link 204  
Spring Data docs  
reference link 148  
Spring Data  
about 32  
URL 32  
Spring Framework  
about 31  
URL 31  
Spring MVC  
about 32  
URL 32  
Spring profiles  
about 178, 179  
reference link 178  
Spring Security  
about 32  
reference link 32  
used, for setting up authorization 145  
SQL databases  
about 39  
H2 39  
MariaDB 39  
MS SQL 40  
MySQL 39  
Oracle 40  
PostgreSQL 40  
stack 9  
Stack Overflow developer survey 2019  
reference link 9  
Swagger CLI 381  
Swagger  
URL 36  
Syntactically Awesome Style Sheets (SASS)  
about 26  
URL 26

## T

technical stack  
about 328, 329, 351, 352  
Bootstrap components, usingreactstrap 333

- HTTP requests, using Axios 333
  - routing, with React Router 331
  - state management, with Redux 330, 331
  - TypeScript, using 329
  - unit testing, setting up 334
  - test-driven development (TDD) 153
  - testing frameworks
    - Cucumber 38
    - Gatling 38
    - JUnit 38
  - testing tools
    - Jest 30
    - Protractor 30
  - tests
    - behavior-driven development (BDD) tests 153
    - client-side end-to-end tests 153
    - client-side unit tests 153
    - fixing 153, 154
    - performance tests 153
    - running 153, 155, 156, 157
    - server-side integration tests 153
    - server-side unit tests 153
  - Thymeleaf
    - about 37
    - URL 37
  - transpilers 12
  - Travis CI
    - about 160
    - URL 160
  - tree shaking 29
  - TypeScript
    - about 28
    - reference link 329
    - using 352
- ## U
- unit testing
    - setup 356
  - use case entity model 98
  - user accounting and authorizing (UAA) 212
- ## V
- View Models (VMs) 73
- virtual machine (VM) 174
  - Vue Router
    - used, for client-side routing 353, 354
  - Vue.js client-side
    - used, for generating application 348, 350
    - used, for generating entity 362, 368, 369
  - Vue.js
    - about 29, 380
    - e-commerce application, creating 373
    - URL 29, 351
  - Vuex
    - reference link 352
    - used, for state management 352
- ## W
- W3C
    - URL 37
  - web application archive (WAR) 14
  - web architecture patterns
    - about 12
    - microservice architecture 16
    - monolithic architecture 13
    - selecting 19
  - webpack dev server
    - reference link 121
  - webpack-serve
    - reference link 121
  - Webpack
    - about 29
    - URL 29
  - WebSocket
    - about 37
    - URL 37
- ## Y
- Yeoman
    - URL 22
- ## Z
- Zipkin
    - about 209, 281
    - URL 209