# Full Stack Development with JHipster

Copyright © 2018 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:** Merint Mathew  
**Acquisition Editor:** Alok Dhuri  
**Content Development Editor:** Akshada Iyer  
**Technical Editor:** Mehul Singh  
**Copy Editor:** Safis Editing  
**Project Coordinator:** Prajakta Naik  
**Proofreader:** Safis Editing  
**Indexer:** Rekha Nair  
**Graphics:** Jisha Chirayil  
**Production Coordinator:**Arvindkumar Gupta

First published: March 2018

Production reference: 1200318

Published by Packt Publishing Ltd.  
Livery Place  
35 Livery Street  
Birmingham  
B3 2PB, UK.

ISBN 978-1-78847-631-7

# Table of Contents

1. [Preface](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/bb6111cc-2da5-45a4-b3ea-c66a85dec10d.xhtml)
   1. [Who this book is for](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7493b236-16f0-460c-ae70-08b6eb24edc3.xhtml)
   2. [What this book covers](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/05e4ee01-3b01-4485-a52b-390d8a1430b9.xhtml)
   3. [To get the most out of this book](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/ccd68f91-56b2-4d75-8662-527810492cdd.xhtml)
      1. [Download the example code files](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/32a88a24-b637-44d1-92dc-dfbb5b1ebd8a.xhtml)
      2. [Conventions used](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/63c211dd-dc96-41e0-9489-5c2919ef97e8.xhtml)
   4. [Get in touch](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/56971e44-b2b3-4895-a43f-f47c7524e9f8.xhtml)
      1. [Reviews](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/c54a33bc-b859-43c4-a85d-8fcfdbf52d8a.xhtml)
2. [Introduction to Modern Web Application Development](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/498dbd6d-b882-4551-92dd-97cdde4b62ac.xhtml)
   1. [Modern full-stack web development](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/36212f7a-826b-44b1-8564-7666a7b27bc4.xhtml)
   2. [Web architecture patterns](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/4ec79e79-c59e-4e6f-8917-05f90d26988c.xhtml)
      1. [Monolithic web architecture](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/e8823a08-e3fb-49e7-97bd-cbfe78961a87.xhtml)
      2. [Microservice architecture](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/0aaf2f9a-1144-40b5-aaad-3125ab9d0a2a.xhtml)
   3. [Choosing the right pattern](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5054a1a9-7f5b-4fbb-8951-4afaa6aaaf4e.xhtml)
      1. [When to choose a monolithic architecture](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/c6b762c0-dce1-47c7-b54b-21e32a23e5e5.xhtml)
      2. [When to choose a microservice architecture](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/9598fa08-03e2-4e57-8f02-26ae3030335e.xhtml)
   4. [Summary](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/1714b671-9af3-4482-b28a-36d29b9eb40f.xhtml)
3. [Getting Started with JHipster](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7fc7275a-b035-4a45-9b55-3a3310572b3a.xhtml)
   1. [Why JHipster?](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3031e1cc-9a02-461c-ab0e-01af90e80f8c.xhtml)
   2. [Goal and adoption of JHipster](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/89ceeb6f-694a-49ec-973b-5e65534a235c.xhtml)
   3. [Introduction to technologies available](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/2a832b05-c35c-4a66-a8f1-7a47770fd32d.xhtml)
      1. [Client-side technologies](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/76466296-a2aa-4f5a-be46-4318ce237524.xhtml)
         1. [HTML5 and CSS3](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f954e839-9d2b-45c0-8528-10b7ef84cbdd.xhtml)
            1. [HTML5](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/d4ba5f66-646b-43f2-9685-856a533c30a6.xhtml)
            2. [CSS3](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/22433202-5f68-48f0-bd1f-edca34556282.xhtml)
         2. [Sass](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/92152067-d0d4-4092-8d24-33f76d768956.xhtml)
         3. [Bootstrap](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/c23ee340-bf6f-437b-a1e6-c4806e5ff86d.xhtml)
         4. [MVVM framework](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/8bc6c5f3-2b21-4d40-9932-5aa2c2c624f1.xhtml)
            1. [Angular](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/9cfd578f-9c29-4bcc-b1d8-a93fc70950ee.xhtml)
            2. [React](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/12ed7fc5-d6ec-498a-955e-fd3d6acd84f3.xhtml)
         5. [Build tools](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/27729208-03a4-4f7e-8200-b77e8933f34b.xhtml)
            1. [Webpack](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/8f135db7-7989-47cf-b755-51fc699a479c.xhtml)
            2. [BrowserSync](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/ad17d446-4168-4cff-85c1-b4e35a9765cb.xhtml)
         6. [Testing tools](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/2987a5be-c791-43c7-84ba-cd360911f7d5.xhtml)
            1. [Karma](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/381709a2-e669-4d0e-8d8c-62d2b0ba489b.xhtml)
            2. [Protractor](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/956bc6a0-a39c-4f53-b601-530ff8624875.xhtml)
         7. [Internationalization](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/87561643-d4ef-4a0a-a4a8-5f70f80df2bc.xhtml)
      2. [Server-side technologies](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7f33bbf6-8c41-4869-bb80-5dd1762c9c30.xhtml)
         1. [Spring Framework](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/1a9765aa-15c8-4f6c-bf7a-d7f93bb5747c.xhtml)
            1. [Spring Boot](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/9b2abe47-602a-4374-b086-16e21c438dee.xhtml)
            2. [Spring Security](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/556230c5-ad2f-4391-bb1a-c9ec6fdc1a14.xhtml)
            3. [Spring MVC](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/eff4dd9e-757e-4fdb-9187-13cb322dcc2b.xhtml)
            4. [Spring data](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/c72edd38-40f2-4dbb-8509-678dca31192d.xhtml)
         2. [Security](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/ee4a7a41-b4a6-4b01-892b-ce2373c98c04.xhtml)
            1. [JWT](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/a504d88a-4958-42c3-8d25-ccea65dd91ba.xhtml)
            2. [Session](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/a998197b-b1c7-4d7d-99cc-be426fd1e2cc.xhtml)
            3. [OAuth2](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/0f9f75d4-cb3d-48ad-9b84-eb64fe572e2b.xhtml)
         3. [Build tools](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/4f4a386e-04f3-4f6f-ae64-640ce81c64d1.xhtml)
            1. [Maven](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/e196e561-9a01-4680-b1c1-da74c20f6966.xhtml)
            2. [Gradle](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/23b04f57-0e26-48b1-95c5-fb1d2b57e569.xhtml)
         4. [Hibernate](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/0c3c8930-a799-4b54-9a49-21913514cd2a.xhtml)
         5. [Liquibase](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/13ac04cb-b952-47bb-aa7c-9407ca9ba660.xhtml)
         6. [Caching](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3d35a0b3-e8c5-433b-bf8e-fe81ac9a5646.xhtml)
            1. [Ehcache](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/ae775c0f-02a1-47ab-876d-32dd09ac504a.xhtml)
            2. [Hazelcast](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5e450a03-768c-4921-9f3e-1be9b391787b.xhtml)
            3. [Infinispan](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3726224d-2b71-43f3-adda-e141cc76182b.xhtml)
         7. [Swagger](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/41bb868c-afc8-4951-8387-754830a0a547.xhtml)
         8. [Thymeleaf](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3281a984-95ee-488e-9f04-e69a1c709ad9.xhtml)
         9. [Dropwizard metrics](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5661a046-02ac-4508-ae00-de370901729a.xhtml)
         10. [WebSocket](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/984e8dcf-eb39-495f-93ec-f9e9325bf87c.xhtml)
         11. [Kafka](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/122971ba-70a0-4765-b512-a5e71d4343eb.xhtml)
         12. [Testing frameworks](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/b6302289-a097-48ae-b3c8-4849b07fa88c.xhtml)
             1. [JUnit](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f94c5ed7-81aa-49b1-b968-743d4016a811.xhtml)
             2. [Gatling](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/0562e1e3-178b-4c84-b034-90db7cc3119e.xhtml)
             3. [Cucumber](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f8b42ba6-13d2-4dde-af1b-40c86390a1b4.xhtml)
      3. [Introduction to database options](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/d45f0622-c794-4aa2-8b7e-06a33908abb7.xhtml)
         1. [SQL databases](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/41a6dd2b-3128-499c-af46-3fdb6b4e0574.xhtml)
            1. [H2](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7f450a2e-42b1-4db1-a318-33ab152bb1ba.xhtml)
            2. [MySQL](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/52304fc9-a6be-4caa-9287-a87d721717c4.xhtml)
            3. [MariaDB](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/07cab9b5-a6af-46d2-ae8e-228b60d00532.xhtml)
            4. [PostgreSQL](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/15cb8428-eade-47eb-a467-d01b62c2bfea.xhtml)
            5. [MS SQL](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/01850823-3f0f-4f60-ac95-b2c58f401400.xhtml)
            6. [Oracle](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/21b6ded2-6ac6-4102-ac36-23793f36aa69.xhtml)
         2. [NoSQL databases](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5180a7bb-22a2-4d5a-9bd8-257ffe1e845a.xhtml)
            1. [MongoDB](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/639eaad9-ceb2-4b9f-9f6a-bea98c1604c6.xhtml)
            2. [Cassandra](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/8e058286-d562-4da3-bcd1-64f4a9b92ddd.xhtml)
            3. [Elasticsearch](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/eba23efd-ce50-4f58-a733-504955cc2a5b.xhtml)
   4. [Installation and setup](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7b6156c1-78b3-4bb8-9053-3e250dcef3aa.xhtml)
      1. [Prerequisites](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f1373b1a-d931-4b3d-90ad-b764e6891c13.xhtml)
         1. [Tools required](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/12f49320-dcc6-4dca-a130-2367d9e481f0.xhtml)
         2. [Installation procedure](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/86ad8334-6d80-4ac9-86ce-79b56128027d.xhtml)
            1. [Java 8](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/b2953e64-0b49-4258-b7ae-092f8adb34b1.xhtml)
            2. [Git](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/991af033-659a-43f0-82ea-1bffad5223b8.xhtml)
            3. [Node.js](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/86e1d257-f1dd-4261-b262-53f1af40b662.xhtml)
            4. [Yarn](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/6545a8b0-5137-4c9f-b342-03a08868e908.xhtml)
            5. [Docker](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/da309f03-9c4b-46b3-839a-fdaeb9cbdeb7.xhtml)
         3. [IDE configuration](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/d1404949-2def-4abb-813e-cbbeff94fb62.xhtml)
         4. [System setup](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/8a5ddd58-169c-4b4e-b8ba-6015a1b71a3a.xhtml)
      2. [Installation of JHipster](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/fadd8838-65fd-4d07-bf27-1c175bd8ca0a.xhtml)
   5. [Summary](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f9ae5ce8-68d6-4ad4-981b-ae9ca39fe184.xhtml)
4. [Building Monolithic Web Applications with JHipster](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3c863445-9d8d-4cd3-b14e-5331593913e3.xhtml)
   1. [Application generation](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3cbb49f7-bc70-4972-93ac-3ccbc20a30ea.xhtml)
      1. [Step 1&#xA0;&#x2013;&#xA0; preparing the workspace](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/24efcb82-5b4a-45fd-bc15-316de4a52113.xhtml)
      2. [Step 2&#xA0;&#x2013; generating code using JHipster](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/325ea918-8145-4e0b-ad47-9fa828bfce43.xhtml)
         1. [Server-side options](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/c19730a5-1c76-4aef-8a70-781ade1eb460.xhtml)
         2. [Client-side options](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f0c7beb1-3c58-49b2-892e-dd90bd6f5d4a.xhtml)
         3. [Internationalization options](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/e9bbff25-ff57-4ec0-aefb-2a6636736232.xhtml)
         4. [Testing](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/4513486b-b1ae-4c93-a50b-9a4d7e2bf106.xhtml)
         5. [Modules](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/d2b6073b-6d5b-4e1c-b602-20889debda03.xhtml)
   2. [Code walkthrough](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/e45425e0-ce33-4ad2-8360-61af7ad51c9a.xhtml)
      1. [File structure](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3be77b2f-4c56-464c-a2ba-b5dcb4eb8776.xhtml)
      2. [Server-side source code](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/a3447835-843b-45b8-95e3-3b9a1cbff9f2.xhtml)
         1. [Java source](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f98a86d2-f77c-495d-b17d-c67a918af2ae.xhtml)
         2. [Resources](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/43077229-dce6-43f0-a044-569efada95f5.xhtml)
      3. [client-side source code](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/92e6d484-adf3-46e8-969a-f495564622e3.xhtml)
   3. [Starting the application](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/d7b47b09-78b1-4e73-9d08-43054910f29c.xhtml)
   4. [Application modules](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/eba1dd8b-fc9c-47e2-b16d-4a223ce5b926.xhtml)
      1. [Home and Login modules](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/9afb5691-f598-4fcb-a817-269e14882a40.xhtml)
      2. [Account modules](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3a3b9bf5-0fea-4da6-8d0d-2f064d22df4f.xhtml)
         1. [Settings](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/8adcad17-d35b-45bb-8fac-90f7f17a2921.xhtml)
         2. [Password](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5ecf9670-0f61-49fe-a9ab-158a43bc4f13.xhtml)
         3. [Registration](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3ae992f7-bf11-4d42-abf4-e7e03f273703.xhtml)
      3. [Admin module](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/229470e3-d4a0-4320-b9b7-c19f3a9ff8c8.xhtml)
         1. [User management](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/1e5b64b2-bdc7-4ecb-8f90-0c05884fc0bd.xhtml)
         2. [Metrics](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/125e60de-42b3-41d4-a32b-19bd042b83b0.xhtml)
         3. [Health](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/da1b8492-ad6a-41f9-93bd-1158e55ef270.xhtml)
         4. [Configuration](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/2ce43268-286a-4ce4-89e0-8928e93232b6.xhtml)
         5. [Audits](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/39e1ec7d-f636-406c-b1ce-07d450712003.xhtml)
         6. [Logs](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/802d8909-4e01-4ab1-a8ae-f2a678a9e558.xhtml)
         7. [API](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/989cde5d-e2de-402d-b08c-1cc165df08fa.xhtml)
   5. [Running generated tests](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/e62f2da9-89b5-4bef-8daf-405f85525aa7.xhtml)
      1. [Server-side tests](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/9ea310df-aed5-4541-b89b-b66db3853c4a.xhtml)
      2. [Client-side tests](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/69a0cd1d-35b4-40f0-887e-59bd81c3b7fa.xhtml)
   6. [Summary](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/d7e9c0fb-7748-4674-9540-727d6a341c2d.xhtml)
5. [Entity Modeling with JHipster Domain Language](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/13a40f14-0d36-43ac-b397-4a09c481c113.xhtml)
   1. [Introduction to JDL](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/35b18dbd-8289-42b6-8061-42e7ec2ab949.xhtml)
      1. [DSL grammar for JDL](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/080198ae-2746-48ba-a402-1107fb028bfa.xhtml)
         1. [Entity modeling with JDL](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/691a6b97-8487-4516-9850-899f5895c12a.xhtml)
         2. [Relationship management](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/14a657a2-a4e7-4ac4-910c-f1e5fe092614.xhtml)
         3. [DTO, service, and pagination options](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/d1663fa8-c1ec-4cc5-b800-8422d7741ab0.xhtml)
      2. [JDL Studio](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/a3653698-a782-47f5-8157-031c37cf6219.xhtml)
      3. [Use case entity model with explanation](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/065342bd-60bd-4c08-ad25-c738f5447ee9.xhtml)
         1. [Entities](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/8d02f0dd-3b7b-4d2a-bbcf-e2fb9378fd36.xhtml)
         2. [Relationships](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/ac7533f1-dac1-45ee-91ce-df93385f439c.xhtml)
         3. [Options for entities](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/300e472e-8e1b-4207-8142-ba63f38dbb41.xhtml)
   2. [Entity generation with JHipster](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f9309b17-7375-41e3-87c8-7099b9df85b6.xhtml)
      1. [Generated code walkthrough](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/babce498-a74c-4ba9-ad5b-1c1304eed775.xhtml)
         1. [Server-side source code](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/68031089-4c61-4a8f-802e-c213d8d5d6f6.xhtml)
            1. [Domain class for the entity](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5892fc44-edf4-4ff1-a84c-1b03710e0c47.xhtml)
            2. [Repository interface for the entity](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/0d36929e-5936-4553-8f56-581530005b72.xhtml)
            3. [Service class for the entity](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/a51a0149-06e9-4ca9-b304-a691a2900a38.xhtml)
            4. [Resource class for the entity](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/ec37fca9-54fc-4180-9e2d-30231ca7f20b.xhtml)
         2. [Client side](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/4e691460-cb6c-4d13-87d1-a1b22323ce22.xhtml)
            1. [TypeScript model class for the entity](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f7589064-c045-4c39-bd4f-c4b10c2e13f6.xhtml)
            2. [Angular services for the entity](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/6c148203-5a57-42e3-8eed-87df09092941.xhtml)
            3. [Angular components of the entity](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/ca143d97-b9a1-46ae-a1a8-7c893d126ed4.xhtml)
            4. [Angular route for the entity](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7d5b9c13-c8e8-4822-9de6-1cf5ffdb50e7.xhtml)
            5. [Angular module for the entity](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/21383ca3-10f0-4c34-83da-36c3724c624a.xhtml)
   3. [Generated pages](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/4979324a-f899-47de-835f-144e934c110c.xhtml)
   4. [Running generated tests](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/9ac5163c-9bdd-4c12-bd25-dd99db37a6cc.xhtml)
   5. [Summary](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3a21d7c7-2c5d-4e10-9487-1ba6304e6f89.xhtml)
6. [Customization and Further Development](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/63e6ed92-1616-40b7-86b8-1b3332fcbfb7.xhtml)
   1. [Live reload for development](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/cbb5fb09-2293-43f4-b0f2-8d27932ebc77.xhtml)
      1. [Spring Boot DevTools](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7fd69929-c0a7-4667-bf0c-473b062d73bf.xhtml)
      2. [Webpack dev server and&#xA0;BrowserSync](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/801bace2-3347-4564-86eb-2e2334cd6cd2.xhtml)
      3. [Setting up live reload for an application](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/c30fb73f-fc77-422f-9edd-11eb7a3d8a9e.xhtml)
   2. [Customizing the Angular frontend for an entity](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7f35872d-e771-4d80-991b-0fc0e39f24f7.xhtml)
   3. [Editing an entity using the&#xA0;JHipster entity sub-generator](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/4a1d3e16-5569-4091-a808-4e4ec0c6a598.xhtml)
   4. [Changing the look and feel of the application](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/2a7e1f03-5b23-46b4-91f3-e320e61cb036.xhtml)
   5. [Adding a new i18n language](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5ef6567e-7cf5-4fb0-8c83-4ec8e52b2278.xhtml)
   6. [Authorization with Spring Security](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/56d41574-3450-4263-896a-3120cecd54fe.xhtml)
      1. [Limiting access to entities](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/494e9add-ba7b-4c16-8ac4-e65cab5024b9.xhtml)
      2. [Limiting access to create/edit/delete entities](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5346f855-e3d0-4cb5-85f0-c1386f2a8a97.xhtml)
      3. [Limiting access to data of other users](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/92c4f625-a208-44f0-be8c-b0c182daaf03.xhtml)
   7. [Summary](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/8b109824-8986-4a8a-9527-ec4e2ad941e7.xhtml)
7. [Testing and Continuous Integration](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/39f6bb2a-8691-42c6-b010-ba71ee36a6c9.xhtml)
   1. [Fixing and running tests](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/b0168a6e-3154-4ed3-9888-5627d99b081b.xhtml)
   2. [Continuous integration](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/6bdd65ad-a575-4e38-966e-93b41f02dce2.xhtml)
   3. [CI/CD tools](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/2aafef23-1cb9-43de-9554-5fdfb12b74f9.xhtml)
      1. [Jenkins](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5af629dc-b884-4a80-854c-4b0705fcbdae.xhtml)
      2. [Travis CI](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/89677a45-dee6-4c3a-946b-2570fd6e591f.xhtml)
      3. [GitLab CI](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f9161a7d-48a2-484f-a45b-638d0fe33e77.xhtml)
      4. [CircleCI&#xA0;](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7233d430-d7e2-4c5e-bb6a-ec043dec97d1.xhtml)
   4. [Setting up Jenkins](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/714641ff-d939-4990-9cab-6c856d076617.xhtml)
   5. [Creating a Jenkins pipeline using JHipster](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/bbc227e7-3cbb-499d-bc19-678b81fc4c03.xhtml)
      1. [The Jenkinsfile and its stages](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/11427284-a9c2-42d8-a7a5-dd1d05e54f2e.xhtml)
      2. [Setting up the Jenkinsfile in a Jenkins server](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5a1f464b-4638-4305-a4b9-0cb009a0041a.xhtml)
   6. [Summary](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/a46892bd-a359-4e5d-989a-33e5d0a5d26a.xhtml)
8. [Going into Production](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/26b69b7e-2044-4130-8cbe-1b080d2649ea.xhtml)
   1. [An Introduction to Docker](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/27c82b0c-1748-4bf3-81a5-2af593c9e84b.xhtml)
      1. [Docker containers](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/d87bca62-7bdb-4feb-a08d-4df6bb60845c.xhtml)
      2. [The Dockerfile](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/69b528ee-f835-4e68-91af-318660c39650.xhtml)
      3. [The Docker Hub](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/050d64a2-4f75-47c2-b1d1-eb7525ab28fc.xhtml)
      4. [Docker compose](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/8b359c5b-a587-4b02-acad-7b801aae6221.xhtml)
   2. [Starting the production database&#xA0;with Docker](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/44adc587-a2e9-48ca-b3eb-50db8ca04d00.xhtml)
   3. [An introduction to Spring profiles](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/cc0a89a3-8373-4978-acca-90001ceb2ec6.xhtml)
   4. [Packaging the application for local deployment](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/995df1fe-d579-4d36-86fa-f392e00fd15d.xhtml)
      1. [Building and deploying using Docker](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7ddaac70-c78f-46ba-b243-4ab91cffb10e.xhtml)
      2. [Building and deploying an executable archive](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3e42224e-2834-435d-a251-79c285588f7c.xhtml)
   5. [Upgrading to the newest version of JHipster](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/63500689-bc2f-49b0-ab52-cc6c2e63cac2.xhtml)
   6. [An introduction to deployment options supported by JHipster](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/e6cc699a-bb55-4377-931c-4af2ae4bb72e.xhtml)
      1. [Heroku](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/c8f391f8-139e-4265-aeeb-322afe9e487a.xhtml)
      2. [Cloud Foundry](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/9bdfba4d-e74e-4c68-b5e8-601e397b9399.xhtml)
      3. [Amazon Web Services](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/d93f1da4-0cd3-4f7a-b6a5-4df92cef1ca5.xhtml)
   7. [Production deployment to Heroku cloud](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/d4f1eaa9-814d-4773-9ab3-fe27a1417eb5.xhtml)
   8. [Summary](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/9bffb89f-f12d-439a-b491-e5cd28dcccea.xhtml)
9. [Introduction to Microservice Server-Side Technologies](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/673932d7-7927-4a12-8c02-86197cc8d3aa.xhtml)
   1. [Microservice applications versus monoliths](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/e001165d-94a8-4f6c-af41-f1efe05c3513.xhtml)
      1. [Building blocks of a microservice architecture](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7311aed2-2a86-4b2a-a32d-8b2f9684931d.xhtml)
         1. [Service registry](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/ed9d56f0-e78a-4e76-9d55-aeb137a64084.xhtml)
         2. [Service discovery](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/4ef9dce5-1563-4b87-988b-9bc0f63ac58a.xhtml)
         3. [Health check](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/0ac57437-b54f-4353-be41-1ba0f4ea1c2a.xhtml)
         4. [Dynamic routing and resiliency](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/8dc619b0-f204-4cf8-b263-585eb44ba9d4.xhtml)
         5. [Security](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/1bba9ca7-988e-4cf7-a33e-07802c16e29d.xhtml)
         6. [Fault tolerance and failover](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/fec59372-aafc-49be-9946-1d4b1bff09c0.xhtml)
      2. [JHipster Registry](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/95ae6d26-cc9d-4c5e-b93c-253ae3aa25a1.xhtml)
         1. [Netflix Eureka server](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/e17890be-8c70-4362-81c3-3afc6e064c41.xhtml)
         2. [Spring cloud config server](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/9a993445-2a6a-465b-8d14-fdc56adceb54.xhtml)
      3. [HashiCorp Consul](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/92a68a7d-0d39-4701-ad37-421050f62bb1.xhtml)
         1. [Service discovery](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/c36fb108-9799-4fc6-9cc4-6642177c1541.xhtml)
         2. [Health discovery](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/4552f252-f103-4af0-8ce1-83444ee3c7fb.xhtml)
         3. [K/V store](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/c25f5962-ff00-46f3-9c85-7b9388b9002e.xhtml)
         4. [Multiple data centers](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/140f12b0-9cd1-4aa6-b5fc-75f7fd065c2b.xhtml)
      4. [JHipster Gateway](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/b6265195-0ecb-4eb4-8627-8bb1392c2e54.xhtml)
         1. [Netflix Zuul](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/76cafcd9-99d5-4eef-936c-2d93b61f8eb5.xhtml)
         2. [Hystrix](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/cb2c41e6-2d2c-40b1-95de-15a6c3f04e13.xhtml)
      5. [JHipster Console](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/786bafec-4831-4ae7-a876-dbb1aeebfb60.xhtml)
         1. [Elasticsearch](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/ea37e6a6-ed0f-4591-8de8-c5824c8b1890.xhtml)
         2. [Logstash](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/4b82a5c3-551c-488e-898e-12415b0295bd.xhtml)
         3. [Kibana](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/c1d32a15-5f8e-4c48-a9fa-b7d952e9a352.xhtml)
         4. [Zipkin](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/8d603dbd-6c80-46d8-a3d3-cfaf813bad2c.xhtml)
      6. [Prometheus](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/723e66ef-5838-4c84-8c18-c337f92312a5.xhtml)
      7. [JHipster UAA server](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/2817d2ba-1a9f-4230-95e6-7dd918d60342.xhtml)
   2. [Summary](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/4b01b5f8-76df-4903-85ce-2768de7bb02e.xhtml)
10. [Building Microservices with JHipster](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/49b2ec94-463c-415c-a2f5-0d4e6c2943b7.xhtml)
    1. [Application architecture](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/ccf347c4-3d6e-4d24-815d-1ac0933b1c62.xhtml)
    2. [Gateway application generation](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7a888698-7712-40af-924e-8af0619d00f9.xhtml)
       1. [Converting a monolithic application to a microservice gateway](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/de0b7fd6-381c-4982-987a-410a5cc4a80e.xhtml)
       2. [Application generation](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/1d1ea052-7f34-40a9-b31d-0833e2010419.xhtml)
          1. [Generating a new Gateway](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5bde0c35-04dc-497d-91ab-24133f2e28cc.xhtml)
       3. [Gateway configuration](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3d13729f-2196-4131-98bd-0e0b9d0ab8cf.xhtml)
       4. [JWT authentication](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/dd14f349-88c4-4df5-9c90-55914647778a.xhtml)
          1. [How JWT works](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f8b0560f-ca0f-4920-ae97-46d1443bc974.xhtml)
    3. [Microservice application - &#xA0;Invoice Service with MySQL database](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5d99f6c2-b3f6-447b-89fe-f9eb5c93d824.xhtml)
       1. [Application generation](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3a85d8a6-467e-4c33-a9ad-6e2f6cf7ac0a.xhtml)
       2. [Microservice configuration](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/cb8d7700-8a83-43d5-82f6-1226b52c0b61.xhtml)
    4. [Microservice application - notification service with NoSQL database](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/98401105-78b9-40b6-b1b7-acaed63ce9d9.xhtml)
       1. [Application generation](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/a08cd59c-8c88-4267-8c08-b536e9c709a5.xhtml)
       2. [Microservice configuration](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f4d3c810-8a52-4ab8-8572-3bafbdae9c1d.xhtml)
    5. [Summary](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3a3da754-19b0-4bdb-a032-7af0d7f9bf04.xhtml)
11. [Working with Microservices](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/28229d3b-052f-436e-9dc9-322727b9be4a.xhtml)
    1. [Setting up JHipster Registry locally](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/71799e7c-59e3-43f8-90ce-0663791d8892.xhtml)
       1. [Using a pre-packaged WAR file](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/152c3264-dc68-41de-afec-507c1e1be089.xhtml)
          1. [Building from source](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7b4efafd-6969-4882-a412-e35d68060ece.xhtml)
          2. [Docker mode](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/be8133cc-06aa-430a-b70b-38949b3bd2b0.xhtml)
    2. [Running a generated application locally](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/0d87c8b4-4e48-48ad-8ef9-cd7687133138.xhtml)
       1. [Gateway application pages](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/2e793800-7918-4540-a247-b9f7f5fc70b6.xhtml)
       2. [JHipster Registry pages](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/b9c59a66-d091-4102-88ab-0ac6e779f283.xhtml)
          1. [System status](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/18e555e6-d6c2-4621-80d6-10b6187dae46.xhtml)
             1. [Below renew threshold](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/99168248-0066-4b14-9796-c3723eb7fc49.xhtml)
          2. [Instances registered](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/e73d4ee8-291c-40ba-b237-2e25da1a75c1.xhtml)
          3. [General info and health](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/82a92496-4690-44ee-a448-09fd863f75f2.xhtml)
          4. [Application listing page](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/20baf3fa-427c-435e-9591-00aa2e0e79c6.xhtml)
          5. [Metrics page](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/aff0228c-3abb-4286-8d47-eedfa87bac18.xhtml)
          6. [Health page](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/697ebd33-8b85-479b-925e-89d7c369ba6c.xhtml)
          7. [Configuration page](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/e49faa3c-3e05-4165-ab47-421e1f8161f5.xhtml)
          8. [Logs page](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/43c73d7f-2a3a-45fe-bd94-531eb628b751.xhtml)
          9. [Swagger API endpoints](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/d844ef51-1ffc-4a66-9322-4f33fa3f45c3.xhtml)
       3. [Running invoice and notification applications locally](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/cc75df4a-87b3-4b50-b683-929ed3d69720.xhtml)
    3. [&#xA0;Modeling entities in JDL&#xA0;](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/4ad2816c-0a94-413e-8a1a-b65d760952a1.xhtml)
    4. [Entity generation on microservices](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/48db352f-8e43-4f01-b79e-af72dbda4d8e.xhtml)
       1. [Explaining the generated&#xA0;&#xA0;code](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/cd8c946d-7ffa-47fa-957a-6c451ed547d3.xhtml)
          1. [Gateway application](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/1b1b7e30-8ec2-454a-a5ce-300c26c0f3fb.xhtml)
       2. [Explaining the generated pages](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/c8851994-6fbf-4518-b5b7-554094aa62b8.xhtml)
    5. [Summary](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/97684b98-b552-487e-a24d-c2f89c0e51e6.xhtml)
12. [Deploying with Docker Compose](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/9362521a-199a-4dda-ae0d-e55d9ed76957.xhtml)
    1. [Introducing microservice deployment options](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/75008fba-0ffe-43a8-800f-74afaf1946ce.xhtml)
       1. [A short introduction to Docker Compose](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f1963f9d-d6ac-4170-90e0-f640a81329d3.xhtml)
       2. [Kickstarting&#xA0;Kubernetes](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/d40b29fa-22c6-481c-b3cf-8a6db7833dee.xhtml)
       3. [Introducing OpenShift](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/19c959ba-6f68-46cb-b9bb-e850e80181ea.xhtml)
       4. [Explaining&#xA0;Rancher](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/48a32663-f693-4db5-a794-ae1f963573ab.xhtml)
    2. [Generated Docker Compose files](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7b979cad-3b9f-40a4-ae57-fc3a843986e7.xhtml)
       1. [Walking through the generated files](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/fa57b9aa-e337-4155-961b-f15666b4828f.xhtml)
       2. [Building and deploying everything to Docker locally](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/40724f6c-a42e-413d-bd16-bb83d900d81e.xhtml)
    3. [Generating docker-compose files for microservices](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/44fbbff7-7a68-47eb-aeec-2fd32facd443.xhtml)
       1. [Features of the deployed application](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/52dbd7f6-141e-41aa-be23-1adac19d6d2f.xhtml)
       2. [JHipster console demo](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/4e0f1c67-fadf-488c-a015-e040203b07b8.xhtml)
       3. [Scaling up with Docker Swarm](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/1cf14eb8-92a8-43f1-aedd-922ab9414fe7.xhtml)
    4. [Summary](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5eb91bb5-86fc-46e5-956e-278edb02fd8f.xhtml)
13. [Deploying to the Cloud with Kubernetes](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7181a834-6940-4e62-be87-3e2f9b3a1ebf.xhtml)
    1. [Generating Kubernetes configuration files with JHipster](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/94b3a37d-a333-4501-902b-69dd3b2b54d9.xhtml)
    2. [Walking through the generated files](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/44299231-4555-43e7-a1f3-6ffb8fbb4bb9.xhtml)
    3. [Deploying the application to Google Cloud with Kubernetes](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/6e48ee0a-ecd4-4a2a-b4c5-65ba87839021.xhtml)
    4. [Summary](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/cec2b250-ea1a-4eae-8ed0-2d9384ebf475.xhtml)
14. [Using React for the Client-Side](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/8c313fad-8f6d-4512-acc1-ae983c210fcf.xhtml)
    1. [Generating an application with React client side](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/4db6bfd0-ae0c-4bb8-b9eb-fc0b892a966f.xhtml)
    2. [Technical stack and source code](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f8941a09-79ec-428a-86e5-081bacc55c8c.xhtml)
       1. [Technical stacks](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3d01c5dd-c5b5-4c06-95e1-84960687e77d.xhtml)
          1. [Using TypeScript](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/14ac5145-261b-40c6-bdcb-3881cf6ba42f.xhtml)
          2. [State management with Redux and friends](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/ddc587d4-86d0-46a7-a93c-fbfc1faeed03.xhtml)
          3. [Routing with React Router](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f319ae15-8b54-4bcd-b0e3-dc22adc2cd88.xhtml)
          4. [HTTP requests using Axios](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5344a090-cee3-4cac-b754-caf03a2f366d.xhtml)
          5. [Bootstrap components using Reactstrap](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/b55d8d36-13a8-44f9-bcb8-43d57f07f42e.xhtml)
          6. [Unit testing setup](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/84a59e00-c6c8-4ba9-aeaa-59939b9929d9.xhtml)
       2. [Generating source code](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/1d2b3af1-4715-4138-a329-fc367d716483.xhtml)
    3. [Generating an&#xA0;entity with React client side](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/b9f36135-015c-47c2-996b-e11a60ae0f6d.xhtml)
    4. [Summary](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/486fd69c-3e43-40a0-a87e-92a15cb997bc.xhtml)
15. [Best Practices with JHipster](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7a7b29a3-0ff7-4d86-be16-2c4b88d3a16a.xhtml)
    1. [The next steps to pursue](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/2ccd72a8-4b24-4c1d-809c-305766aa84f3.xhtml)
       1. [Adding a shopping cart for the application](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/bd632141-0ff1-47a9-bc9f-3913e38f1fed.xhtml)
       2. [Improving end-to-end tests](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/e8f4fb16-3482-4a68-a5d1-e49aa763ad41.xhtml)
       3. [Improving the CI/CD pipeline](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/0765ea32-de89-49f1-821f-3813a1f9dd57.xhtml)
       4. [Building a JHipster module](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/5a30d3dd-ee74-45fb-8b62-9a73a1b28bcf.xhtml)
    2. [Best practices to keep in mind](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/1848f32a-0d5d-4b2b-a5b5-ab81dc3cd947.xhtml)
       1. [Choosing a client-side framework](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/c94cb458-7f86-44ff-94cd-94e650053b30.xhtml)
       2. [Choosing a database option](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7c8747be-c486-480e-a81f-fc1fe04533e5.xhtml)
       3. [Architecture&#xA0;considerations](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/a2511b85-9fb1-465e-86e9-9f87e0cbbe0c.xhtml)
       4. [Security considerations](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/6cf4aaca-8a24-469d-9129-6f1f53d80efb.xhtml)
       5. [Deployment and maintenance](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7528593f-7105-4b46-8606-0e2e6080c214.xhtml)
       6. [General best practices](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/f84149c5-b4b7-415f-8a42-ce7c441452c5.xhtml)
    3. [Using JHipster modules](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/8ee54939-d451-48bd-a842-fb0e2c085a1d.xhtml)
    4. [Contributing to JHipster](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/453a55cd-2cb9-4105-89cd-639d7ee80554.xhtml)
    5. [Summary](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3ac465c2-b54f-46d8-9ae6-2f959f511316.xhtml)
16. [Other Books You May Enjoy](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/0a684b4e-d2e7-4443-8731-f8a08e8789ad.xhtml)
    1. [Leave a review - let other readers know what you think](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/9010511d-dca0-497a-95d7-10b8b009e760.xhtml)

**Preface**

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

* There are multitudes 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 together. 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 will go on to develop and deploy features and business logic on cloud services. You will also learn about microservices and how to convert a monolithic application into the microservice architecture as it evolves using JHipster. Additionally, you will learn how to make use of the new React support being introduced 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 can 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 or React
* 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

# What this book covers

[Chapter 1](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/498dbd6d-b882-4551-92dd-97cdde4b62ac.xhtml), 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](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7fc7275a-b035-4a45-9b55-3a3310572b3a.xhtml), Getting Started with JHipster, introduces the JHipster platform. It will also give the reader a brief overview of different server-side, client-side, and DB technology options offered by JHipster. This chapter will also provide instructions to install and use JHipster and various tools and options supported by it.

[Chapter 3](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/3c863445-9d8d-4cd3-b14e-5331593913e3.xhtml), Building Monolithic Web Applications with JHipster, guides the user through the creation of a 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](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/13a40f14-0d36-43ac-b397-4a09c481c113.xhtml), Entity Modeling with JHipster Domain Language, introduces the reader to JHipster domain language (JDL) and will teach build business logic with entity modeling and entity creation using JDL and JDL studio.

[Chapter 5](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/63e6ed92-1616-40b7-86b8-1b3332fcbfb7.xhtml), Customization and Further Development, guides the reader through further development of the generated application. It will also teach how to the reader more about using technologies such as Angular, Bootstrap, Spring Security, Spring MVC REST, and Spring Data.

[Chapter 6](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/39f6bb2a-8691-42c6-b010-ba71ee36a6c9.xhtml), Testing and Continuous Integration, guides the reader through testing and setting up a continuous integration pipeline using Jenkins.

[Chapter 7](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/26b69b7e-2044-4130-8cbe-1b080d2649ea.xhtml), 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](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/673932d7-7927-4a12-8c02-86197cc8d3aa.xhtml), Introduction to Microservice Server-Side Technologies, gives an overview of different options available in the JHipster microservice stack.

[Chapter 9](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/49b2ec94-463c-415c-a2f5-0d4e6c2943b7.xhtml), 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 the generated code and components such as JHipster registry, JHipster console, API gateway, and JWT.

[Chapter 10](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/28229d3b-052f-436e-9dc9-322727b9be4a.xhtml), Working with Microservices, guides the reader through running the generated applications locally and creating domain entities for the microservice architecture using JHipster domain language.

[Chapter 11](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/9362521a-199a-4dda-ae0d-e55d9ed76957.xhtml), 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](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7181a834-6940-4e62-be87-3e2f9b3a1ebf.xhtml), Deploying to the Cloud with Kubernetes, guides the user through the Google cloud deployment of the generated microservice stack using Kubernetes and JHipster.

[Chapter 13](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/8c313fad-8f6d-4512-acc1-ae983c210fcf.xhtml), 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](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7a7b29a3-0ff7-4d86-be16-2c4b88d3a16a.xhtml), 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 basics of the following technologies:

* Web technologies (HTML, JavaScript, and CSS)
* Java 8
* Basics of the Spring Framework
* Basic understanding of SQL databases
* Build tools (Maven or Gradle)
* npm or Yarn

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

You will also need JDK8, Git, Docker, and NodeJS 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](http://www.packtpub.com). If you purchased this book elsewhere, you can visit [www.packtpub.com/support](http://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](http://www.packtpub.com/support).
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
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**](https://github.com/PacktPublishing/Full-Stack-Development-with-JHipster). 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/**](https://github.com/PacktPublishing/). Check them out!

# 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: "At 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 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.

**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 [www.packtpub.com/submit-errata](http://www.packtpub.com/submit-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 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](https://www.packtpub.com/).

**Introduction to Modern Web Application Development**

According to the Stack Overflow developer survey 2017 (<https://insights.stackoverflow.com/survey/2017#developer-profile-specific-developer-types>), *full-stack web 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 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 or **client-side** and **server-side** stack. Frontend generally refers to the part that is responsible for rendering the user interface, and backend refers to the part that is responsible for the business logic, database interactions, user authentication, server configuration, 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 as required.

With an ever-evolving software architecture landscape, the scope of technologies that a full-stack web developer is expected to work has increased tremendously. 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, VueJS, and so on. 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 so on.

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**

If we were to even begin discussing the life of a full-stack developer, it would be worthy of a whole book by itself – so let's leave that for another day.

Let's look at a user story about 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 the all the code hence we won't detail them out here:

* 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.
* You will be modeling the domain model for the feature depending on the database technology used.
* Then, you would create server-side code and database queries to persist and fetch data from the database.
* Once the data is ready you would implement server-side code for any business logic.
* Then, you would implement an API that can be used to provide data for the presentation over an HTTP connection.
* You would write integration tests for the API.
* Now, since the backend is ready, you would start writing frontend code in JavaScript or a similar technology.
* You would write client-side services to fetch data from the backend API.
* You would write client-side components to display the data on a web page.
* You would build the page and style it as per the design provided.
* You would write automated end to end tests for the web page.
* It is not done yet. Once you have tested everything works locally you would create pull requests or check-in the code to the version control system used.
* You would wait for the continuous integration process to verify everything, and fix anything that is broken.
* Once everything is green and the code is accepted, typically you would start the deployment of this feature to a staging or acceptance environment, either on-premises or to a cloud provider. If it is 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.
* Once the feature is accepted you might be responsible for deploying it into the production environment in a similar way, and troubleshoot 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 co-worker while s/he deploys yours.
* You might also be responsible, along with your co-workers, to make sure 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 spawns across 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, 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 in keeping yourself up to date with various technologies and patterns in multiple disciplines of software development. 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 revolutionalized the software industry but they also introduced a lot of new stuff to learn and keep track of, such as orchestration tools, container management tools, and so on.
* Cloud services are growing day by day. To stay on track you would have to familiarize yourself with their API 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, Kotlin, and so on, forcing you to keep yourself up to date with them. On the other side, server-side frameworks are becoming more feature rich and hence more complex.

The most important thing of all is the pain of making sure all of these work together well when required. It 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 see the details 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—**monolithic architecture** and **microservice architecture**, the latter being the new kid on the block.

Let's take a look at the following in detail:

* Monolithic architecture
* Microservice architecture

**Monolithic web architecture**

A monolithic architecture is the most used pattern for web applications due to its simplicity in development and deployment. Though the actual moving parts will differ from application to application, the general pattern remains the same. In general, a monolithic web application may do the following:

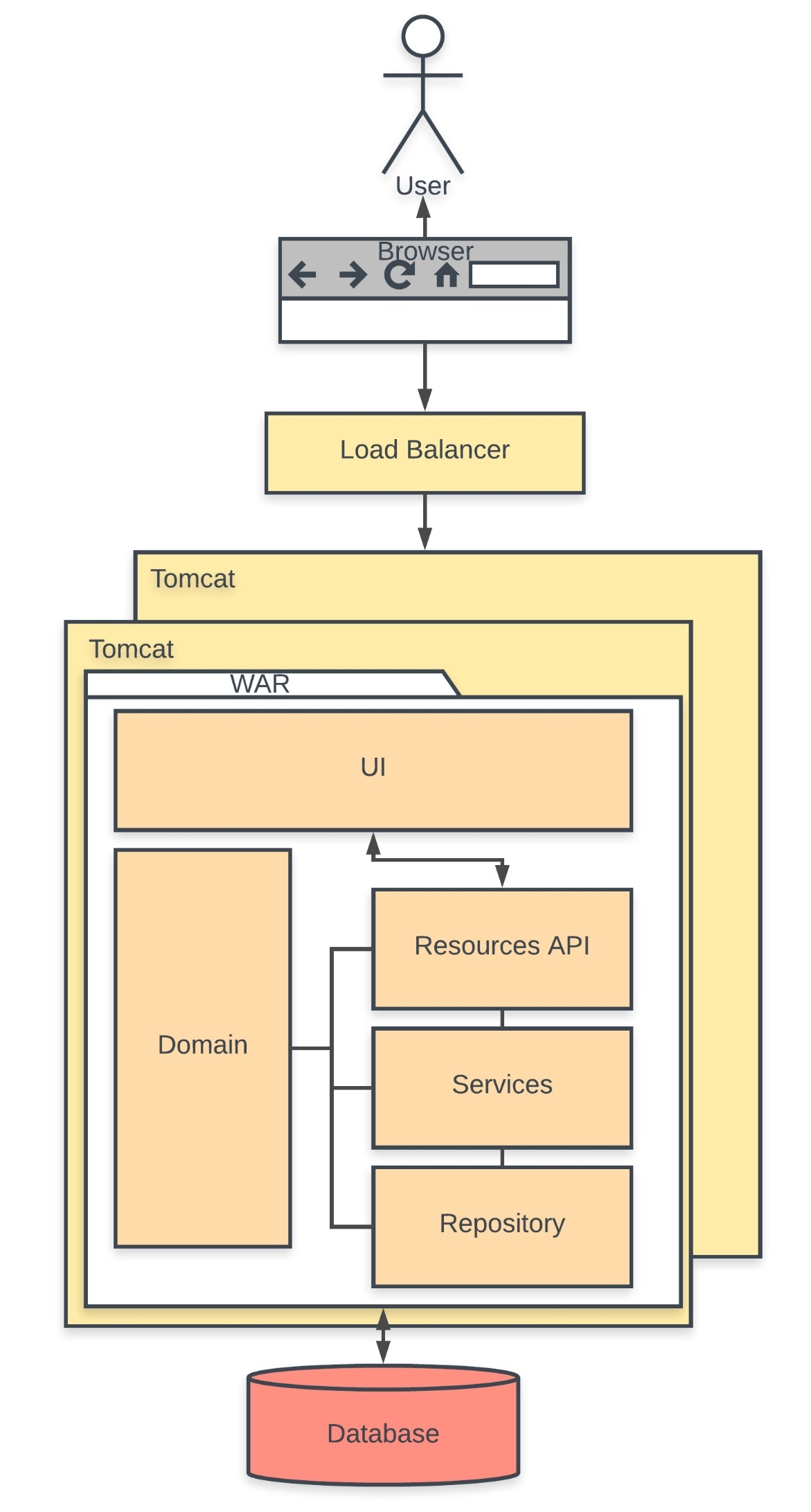
* It can support different clients such as desktop/mobile browsers and native desktop/mobile applications
* It can expose APIs for third-party consumption
* It can integrate with other applications over REST/SOAP web services or message queues
* It can handle HTTP requests, execute business logic, access a database, and can exchange data with other systems
* It can run on web application containers such as Tomcat, JBoss, and so on
* 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** (**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 server or between two systems.   
  
**SOAP** (**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 reservation orders online 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 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 detailed here:

* Simpler to develop as the technology stack is uniform throughout all layers.
* Simpler to test as the entire application is bundled in a single package making it easier to run integration and end-to-end tests.
* Simpler and faster to deploy, as you only have one package to worry about.
* Simpler to scale as you can multiply the number of instances behind a load balancer to scale out.
* Requires a smaller team to maintain the application.
* Team members share more or less the same skill set.
* The technical stack is simpler and most of the times easier to learn.
* Initial development is faster hence making time to market faster.
* Requires 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 detailed here:

* Components are tightly coupled together resulting in unwanted side effects such as changes to one component causing a regression in another and so on.
* Becomes complex and huge over time resulting in slow development turnaround. New features will take more time to develop and refactoring of existing features will be more difficult due to tight coupling.
* The entire application needs to be redeployed for any changes.
* Is less reliable due to tightly coupled modules. A small issue in a service might break the entire application.
* Newer technology adoption is difficult as entire application needs to be migrated. Incremental migration is not possible most of the time. Hence many monolithic applications end up having an outdated technology stack.
* Critical services cannot be scaled individually resulting in increased resource usage as the entire application will need to be scaled.
* Huge monolith applications will have a higher start-up time and high resource usage in terms of CPU and memory.
* Teams will be more interdependent and it will be challenging to scale the teams.

**Microservice architecture**

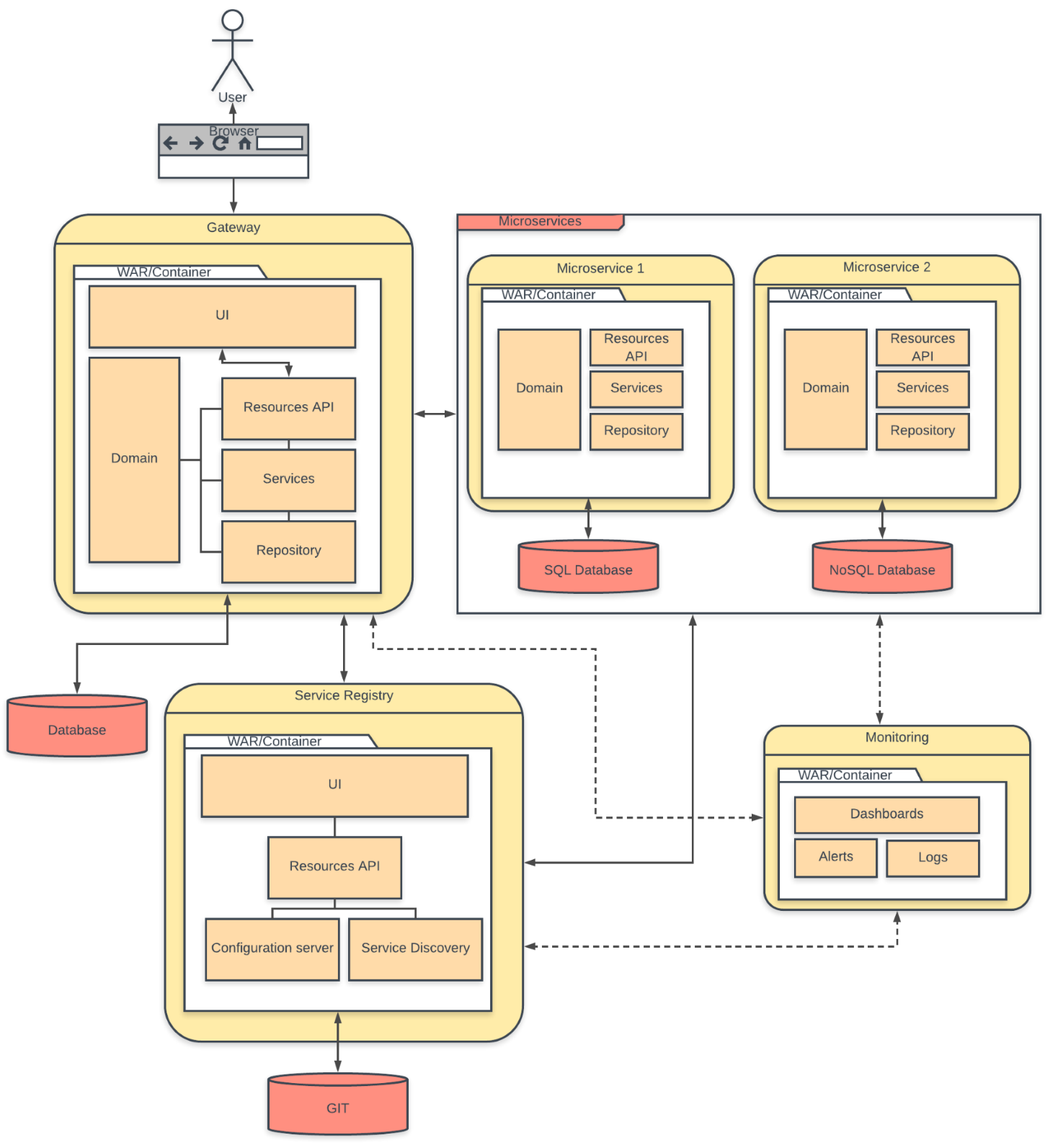
The microservice architecture has gained momentum in recent years, and is gaining popularity in web application development due to its modularity and scalability. Microservice architecture can offer almost all the features of a monolith that we saw in the earlier section. Additionally, it offers many more features and flexibility, and hence is often considered a superior choice for complex 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 share some common traits and they are, in general, the following:

* 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 a single component can choose its own technology stack and programming language.
* They often utilize advanced features such as service discovery, circuit breaking, load balancing, and so on.
* Microservice components are mostly lightweight and they do a specific functionality. For example, an authentication service will only care about authenticating a user into the system.
* 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, maintain 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 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 detailed here:

* Loosely coupled components resulting in better isolation, easier to test and faster to startup.
* Faster development turnaround and better time to market. New features can be built faster and existing features can be easily refactored.
* Services can be deployed independently making the application more reliable and make patching easier.
* Issues, such as a memory leak in one of the services, are isolated and hence will not bring down the entire application.
* Technology adoption is easier, components can be independently upgraded in 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 start-up.
* Teams will be less dependent on each other. Best suited for agile teams.

The disadvantages of a microservice web application architecture are as detailed here:

* 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.
* 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 and virtualization involved.
* Scaling is more efficient but setting upscaling is more complex as it would require advanced features such as service discovery, DNS routing, and so on.
* 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.
* The technical stack is complex and most of the times harder to learn.
* Initial development time will be higher making time to market slower.
* Requires a complex infrastructure. Most often will require containers (Docker) and multiple JVM or app containers to run on.

# Choosing the right pattern

When starting a new project, it is always difficult to choose an architecture pattern these days. There are so many factors to take into account and it is easy to get confused with all the **hype** around different patterns and technologies (see **Hype Driven Development** (<https://blog.daftcode.pl/hype-driven-development-3469fc2e9b22>)). 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 gives an idea of when to go with a monolithic architecture over microservices:

* 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, a simple CRUD application, and so on.
* When the **team size** is small, say less than eight people (it's not a hard limit but rather practical).
* When the **average skill set** of the team is either novice or intermediate.
* When **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 them to grow. For example, an enterprise app targeting a specific set of users.

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

**When to choose a microservice architecture**

The following list can be used as a general guide when choosing a microservice architecture. This is not a definitive list but gives an idea of when to go with microservices architecture over a monolith. Please note that unlike choosing a monolithic architecture, the decision here is more complex and may involve cross consideration among 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, an API provider, and so on.
* When the **team size** is large, there must be 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 **time to market** is not critical. The microservice architecture will take more time to get right up front.
* 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 them to grow. For example, a social media application targeting users all over the world.

Though a monolithic architecture would suffice in most cases, investing up front in a microservice architecture will reap long-term benefits when the application grows huge.

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

# Summary

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

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.

**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 microservice architectures in a jiffy. JHipster also offers various tools to develop the application further using business entities, and deploy it to various cloud services and platforms. At its core, JHipster is a Yeoman generator that creates Spring Boot and Angular/React based applications. It can create monolithic architecture as well as microservice architecture with every feature working out-of-the-box.

In this chapter, we will cover the following topics:

* Why use JHipster and how it helps compared to traditional development approaches
* What is 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 us 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, monitoring, and so on. Finally, you have to test and deploy the application to a cloud service.

If you are approaching this challenge the traditional way you will most probably be doing the following steps. Let's skip the details for simplicity. So, the steps would be 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, test the application, and deploy it

While this approach definitely works, for this simple application you would have spent anywhere between four to six 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.

**Goal and adoption of JHipster**

The goal of JHipster is to provide developers a platform where you can focus on your business logic rather than worrying about wiring different technologies together, and also that provides a great developer experience. Of course, you can use available boilerplate within your organization or from the internet and try to wire them up together, but then you will be wasting a lot of time re-inventing the wheel. With JHipster, you will create a modern web application or microservice 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 or React supported by Bootstrap
* A battle-tested microservice architecture unifying Netflix OSS, Elastic stack, and Docker
* A great tooling and development workflow using Maven/Gradle, webpack, and Yarn/NPM
* Out-of-the-box continuous integration using Jenkins, Travis, or GitLab
* Excellent Docker support and support for orchestration tools such as Kubernetes, Rancher, and Openshift out-of-the-box
* Out-of-the-box support for various cloud deployments
* Above all, great code with lots of best practices and industry standards at your fingertips

Netflix OSS (<https://netflix.github.io>) is a collection of open source tools and software produced by the NETFLIX, INC team geared toward microservice architecture. Elastic stack (<https://www.elastic.co/products>)(formerly known as ELK stack) is a collection of software tools, which help in monitoring and analytics of microservices developed by the Elasticsearch (<https://www.elastic.co>) team.

JHipster has been steadily increasing in popularity as Spring Boot and Angular 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 (beginning of 2018), there are more than 5,000 applications generated per month and JHipster was installed around  1 million times. It has more than 400 contributors with official contributions from Google, RedHat, Heroku, and so on.

**Introduction to technologies available**

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 us 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

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.

# 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

HTML5 (<https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5>) is the latest of the **HTML** (**HyperText Markup Language**) 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, WebGL, and more. JHipster also uses best practices from the 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

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>) that can be used instead of the box model used traditionally. This allows having 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.

# 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 easier 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>) so that pure Angular 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.

# 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 and React as the client-side framework.

# Angular

AngularJS (<https://angularjs.org>)(version 1.x) is a client-side MVVM framework, maintained by Google, which helps to develop **Single Page Applications**(**SPA**). It is based on a declarative programming model and it extends standard HTML with the ability to add additional behavior, elements, and attributes through directives.

Angular (<https://angular.io>)(version 2 and above) is a complete rewrite of the framework and hence is not backward compatible with AngularJS. 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, factory, and so on. 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 **AOT**(**Ahead of Time Compilation**), lazy loading, reactive programming, and so on.

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, class attribute visibility modifiers, and so on. 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, spread operators, and so on.

# 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 JS 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 like React Router (<https://reacttraining.com/react-router>), Redux (<https://redux.js.org>), MobX (<https://mobx.js.org>), and so on. JHipster uses React along with Redux and React Router and similar to Angular, JHipster uses TypeScript for 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.

If you are starting a new project, it is best to choose either Angular or React as they are well maintained. However, with older versions of JHipster, AngularJS 1.x was also offered as an option but it is becoming legacy and will soon be discontinued in JHipster 5.x. JHipster will provide an official blueprint for those who are still interested in using AngularJS 1.x. Just run the command jhipster --blueprint generator-jhipster-angularjs to use it.

# Build tools

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 would 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 and React.

# Webpack

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 JS, 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.

# BrowserSync

BrowserSync (<https://browsersync.io>)is a NodeJS tool that helps in 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, scrolling, and so on. It integrates with Webpack/GulpJS to provide a productive development setup. It makes testing a web page on multiple browsers and devices super easy.

# Testing tools

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 so on. JHipster creates unit tests for client-side code using Karma and Jasmine out-of-the-box and also supports creating end-to-end tests using Protractor.

# Karma

Karma (<https://karma-runner.github.io/2.0/index.html>) is a test runner that can execute JavaScript code in real browsers. It creates a web server and executes the test code against the source code. Karma supports multiple testing frameworks such as Jasmine, Mocha, and Qunit, and integrates well with continuous integration tools.

# Protractor

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, VueJS, and so on. Protractor runs e2e 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

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, for example. 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.

# Spring Framework

The 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 JavaEE vendors before the rise of Spring and soon after Spring, it became the number one choice for Java web developers, giving JavaEE 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 together in a consistent way. The framework has numerous modules aimed at different tasks, such as data management, security, REST, web services, and so on. 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.

# Spring Boot

Spring Boot (<https://projects.spring.io/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 auto-configures Spring wherever possible 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://projects.spring.io/spring-security>) is the de facto solution for security in a Spring Framework-based application. It provides API and utilities to manage all aspects of security, such as authentication and authorization. It supports a wide range of authentication mechanism such as OAuth2, JWT, Session (Web form), LDAP, **SSO** (**Single Sign-On**) servers, **JAAS** (**Java Authentication and Authorization Service**), Kerberos, and so on. It also has features such as remember me, concurrent session, and so on.

# 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 (<http://projects.spring.io/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, custom object mapping abstractions, and so on. 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.

# 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 protect 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 9](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/49b2ec94-463c-415c-a2f5-0d4e6c2943b7.xhtml), Building Microservices with JHipster.

# 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 DB, and can be invalidated, used in remember me mechanisms, and so on.

# OAuth2

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 a service, typically an OAuth2 server. OAuth2 is more complicated to set up when compared to the previously mentioned mechanisms. JHipster supports setting up OAuth 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

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

# Maven

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 itself. Maven has a long history and is much more stable and reliable compared to Gradle. It also has a huge ecosystem of plugins.

# Gradle

Gradle (<https://gradle.org>) is a build automation tool which uses a Groovy DSL to specify the build plan and dependencies. It is a strong contender rapidly gaining popularity and adoption. 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.

# Hibernate

Hibernate (<http://hibernate.org>) is the most popular **ORM**(**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**(**Java Persistence API**) and is the go-to provider for a JPA implementation. Hibernate also offers many additional features such as entity auditing, bean validation, and so on. 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, and eliminate the need for a DBA for most DB management. 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 2nd level cache, and also with Spring Cache abstraction to enable caching at the method level. JHipster supports JCache-compatible Hibernate 2nd level cache provided by EhCache, Hazelcast, and Infinispan.

# 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 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 becomes 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 more features such as in-memory data grids, MapReduce support, and so on.

# Swagger

OpenAPI specification (previously known as **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 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 generated web pages on the server side, for templating email messages and so on. Although server-side web page templates are slowly losing out to client-side MVVM frameworks, it is still a useful tool if one wants to have something more than a single page application using Angular.

# Dropwizard metrics

Dropwizard metrics (<http://metrics.dropwizard.io/4.0.0/>) is an excellent open source library for measuring the performance of your Java web application. Paired with Spring Boot, this can bring a lot of value by measuring the performance of the REST API, measuring the performance of cache layer and database, and so on. Dropwizard provides handy annotations to mark methods to be monitored. It supports counters, timers, and so on.

# WebSocket

WebSocket (<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 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 pushing data from server to client in real-time and for implementations such as real time chat, notifications, and so on. 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 scale has 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 (Java Message Service) (<https://en.wikipedia.org/wiki/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(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

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

# JUnit

JUnit (<https://junit.org/junit5/>) is the most widely used Java testing framework. It is a 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

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 spec. 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

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 looks similar to plain English.

**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** (**RDBMS**) 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.

# H2

H2 (<http://www.h2database.com/html/main.html>) is a free embedded RDBMS commonly used for development and testing. It normally can run in file system 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 normally is 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, partitioning, and so on. It is one of the most preferred 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 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 use it as a NoSQL database or in Hybrid mode. It also has enterprise-ready features such as replication, high availability, and so on.

# MS SQL

MS SQL server (<https://www.microsoft.com/nl-nl/sql-server/sql-server-2017>) 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 used database due to its legacy and enterprise features. It is commercial software and requires a paid license to use. It has enterprise-ready features such as sharding, replication, high availability, and so on.

# 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, content stores, and so on. 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, replication, and so on. It is free and open source software.

MapReduceis 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 faster.

# Cassandra

Apache Cassandra (<http://cassandra.apache.org/>) is 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 is the most popular choice for critical high availability systems. It was originally developed and open sourced by Facebook.

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

# 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 used search engines.

# Installation and setup

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.

# Prerequisites

Before we install the JHipster CLI, let's take a look at the prerequisites. We will need to install some dependencies and configure our favorite IDE to work best with 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 follow, these 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

# Installation procedure

Let us see the installation procedure for each of the tools.

**Java 8**

Java 9 is the latest Java release introducing features like modules, reactive streams and so on. While JHipster applications will work with Java 9 it is recommended to stick to the more stable Java 8 until Java 9 support is stable in all the dependencies used.

The generated applications use Java 8 and hence it is required to compile the applications:

1. Check for your installed Java version by running the command java -version in the Terminal. It should display java version "1.8.x" where x could be any patch version.
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 8.
3. Once installed, check the command in step 1 again to make sure. As the JDK alters the environment variable to set JAVA\_HOME you would have to open a new Terminal here.

**Git**

Git is the most 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 NodeJS 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 can 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, check 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/2017#technology-programming-languages>). The Node ecosystem is the largest in the world with over 600,000 packages and is managed by NPM, the default package manager.

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

1. Check for NodeJS by typing node -v in the Terminal. It should display a version number. Make sure that the version number is greater than 8.9 and corresponds to the latest LTS version of NodeJS.
2. If the command is not found or if you have a lower version of NodeJS then you can visit the Node.js website (<https://nodejs.org/en/download/>) and follow the instructions to install the latest LTS version available. 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 NodeJS alters the environment variables, you would have to open a new Terminal here.
4. NPM is automatically installed when you install NodeJS. You can check this by running npm -v in the Terminal.

You can install multiple NPM packages by running the command npm -g install bower gulp-cli CLI or using Yarn, yarn global add bower gulp-cli.

**Yarn**

Yarn is a package manager for NodeJS. It is API and feature compatible with NPM and provides better performance and a flat package tree.

JHipster, by default, uses Yarn instead of NPM as Yarn is much faster at the time of writing. If you prefer to use NPM, then you can skip this step:

1. You can visit the Yarn website (<https://yarnpkg.com/en/docs/install>) and follow the instructions to install Yarn.
2. Once installed, check by running yarn --version to make sure.

**Docker**

Docker is the defacto standard for container management and it made 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>) for you, based on your choice of build tool. If you don't want to use those wrappers, go to the official Maven website (<http://maven.apache.org/>) or 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 development of a generated application it is highly recommended that you use a proper Java IDE such as IntelliJ, Eclipse, or Netbeans. Sometimes you could 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 folders node\_modules, git, build, and target 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 one might encounter:

* When using Yarn on macOS or Linux, you need to have $HOME/.config/yarn/global/node\_modules/.bin in the path. This will normally be automatically done when you install Yarn but if not, you can run the command export PATH="$PATH:`yarn global bin`:$HOME/.config/yarn/global/node\_modules/.bin" in a Terminal to do this.
* If you are behind a corporate proxy, you will have to bypass it for NPM, Bower, 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.

**Installation of JHipster**

OK, now let's get started for real. JHipster can be used from a local installation with NPM or Yarn, from a Vagrant image provided by the team, 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 would be by installing the JHipster CLI using Yarn or NPM. Open a Terminal and run:

**> yarn add global generator-jhipster**

If you would prefer NPM, then just run:

**> 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:

https://learning.oreilly.com/library/view/full-stack-development/9781788476317/assets/ed5d567c-fccc-4564-b9d8-c115205405d6.png

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 following the preceding steps:

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

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 Vagrant development box or the Docker image from the JHipster team. Visit <https://github.com/jhipster/jhipster-devbox> for instructions to use the Vagrant box or visit <http://www.jhipster.tech/installation> and scroll down to the Docker installation (for advanced users only) section for instructions to use a Docker image.

# Summary

In this chapter, we discovered JHipster and the different technology options provided by it. We had a brief introduction of the important pieces of the client-side and server-side stack. We had a quick overview of Spring technologies, Angular, Bootstrap, and so on. We also had an overview of different database options supported by JHipster. We learned about the tools required to work with JHipster and we have successfully set up our environment to work with JHipster and installed JHipster CLI. In the next chapter, we will see how JHipster can be used to build a production-grade monolithic web application.

**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 front end. 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. For this exercise, we will only be building a simple UI for the client-facing part. We will talk about other option as we go through this chapter.

In this chapter, we will:

* See how to create a monolithic web application using JHipster
* Walk through important aspects of the generated code
* See the security aspects of the generated application
* See how to run the application and tests
* See the generated frontend screens
* See the tools included that will ease further development

This chapter will require the use of a terminal (command prompt on windows) app throughout. You can the see previous chapter for more info about that.

# Application generation

Before we start generating the application, we need to prepare our workspace as this workspace will be used throughout this book, and 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. Let's first 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 the tools as they might include important bug fixes. You can upgrade JHipster anytime using the command yarn global upgrade generator-jhipster

**Step 2 – generating code using JHipster**

Initialize JHipster by running the jhipster command into the terminal, which will produce the following output:



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

* **Monolithic application**: As the name suggests, it creates a monolithic web application with a Spring Boot-based backend and an 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 towards a microservice architecture with additional configurations. It features an 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.

We will choose the **monolithic application** for our use case. We will talk and look at the other options in detail in [Chapter 8](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/673932d7-7927-4a12-8c02-86197cc8d3aa.xhtml), *Introduction to Microservice Server-Side Technologies*, of this book.

Run jhipster --help to see all 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.

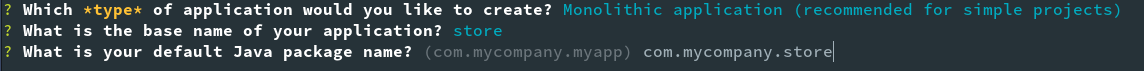
**Server-side options**

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 for creating the main class file names, database names, and so on. By default, JHipster will suggest the current directory name if it doesn't contain any special characters in the name. Let's name our application as store. Please note that the files will be created in the current directory you are in:



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

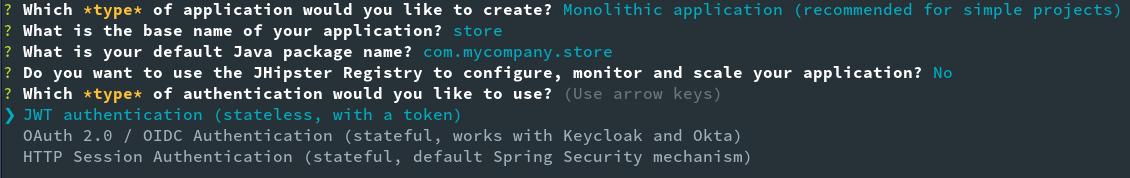


* **Question 3**. This prompt asks whether we need to configure JHipster registry for this instance. JHipster registry provides a service discovery and config server implementation which is very useful for centralized configuration management and scaling of the application. For this use case, we will not need it, so let's choose No. We will learn more about the JHipster Registry in [Chapter 8](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/673932d7-7927-4a12-8c02-86197cc8d3aa.xhtml), *Introduction to Microservice Server-Side Technologies*, of this book:

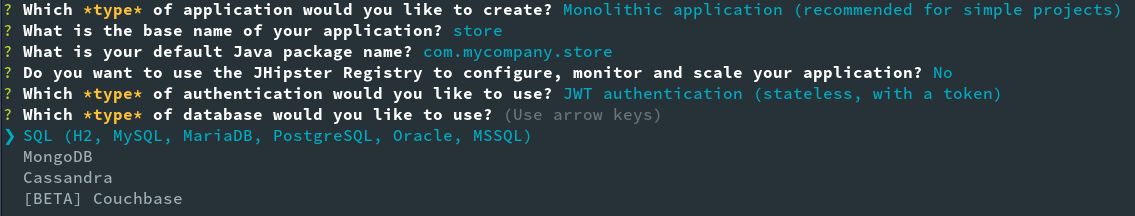


* **Question 4**: This prompt asks us to select an authentication mechanism. We are presented with three options:
  + JWT authentication
  + HTTP Session Authentication
  + OAuth 2.0/OIDC Authentication

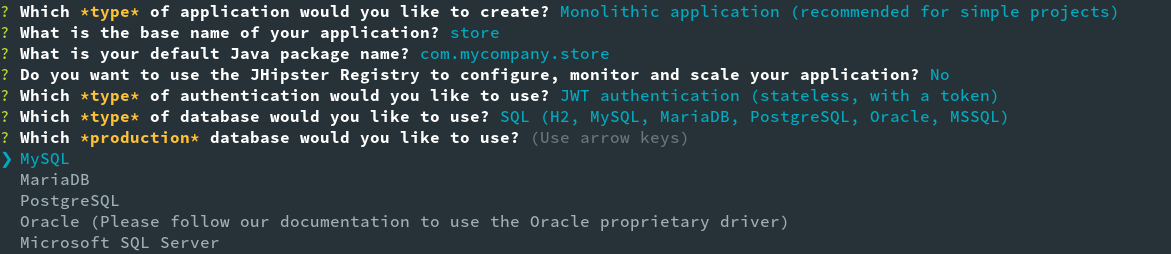
We already saw how these defer in the previous chapter, and for our use case, let's choose JWT authentication:



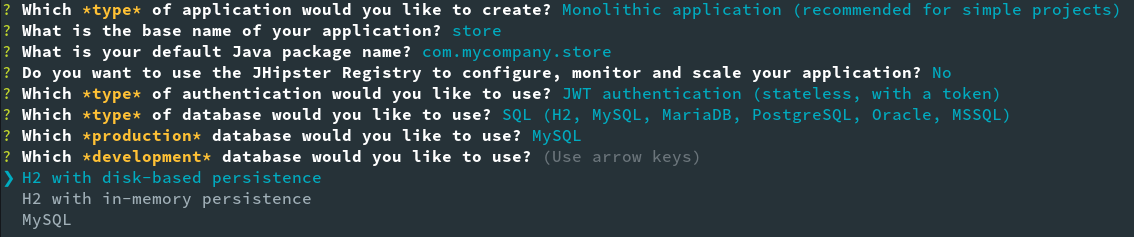
* **Question 5**: This prompt asks us to select a database type; the options provided are SQL, MongoDB, Couchbase, and Cassandra. We already learned about the different database options in the previous chapter. For our application, let's choose an SQL database:



* **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 here:

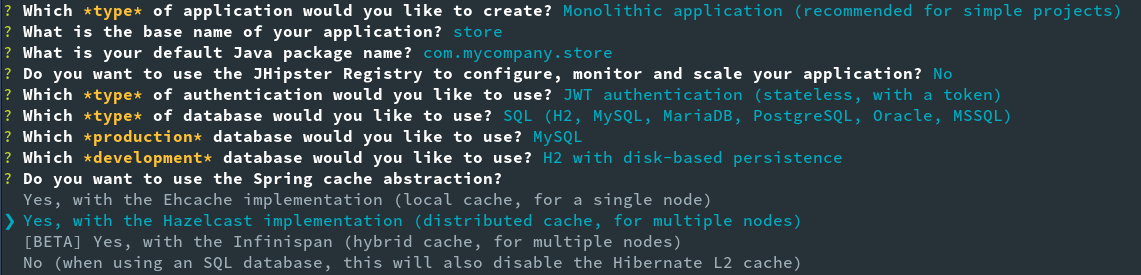


* **Question 7**:This prompt asks us to choose between our chosen SQL database and H2 embedded database for development. H2 embedded DB is especially useful as it makes development faster and self-contained, without the need to have a MySQL instance running. So, let's choose the H2 disk-based persistence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

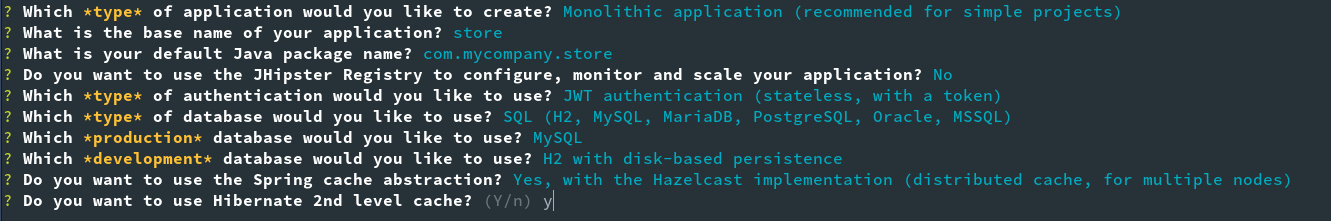


If your use case requires working with persisted data in development and if the model is not going to change often, then you could also choose MySQL for development as it would give you a faster startup time. This is because the embedded H2 DB doesn't need to be initialized, but the downside is each 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.

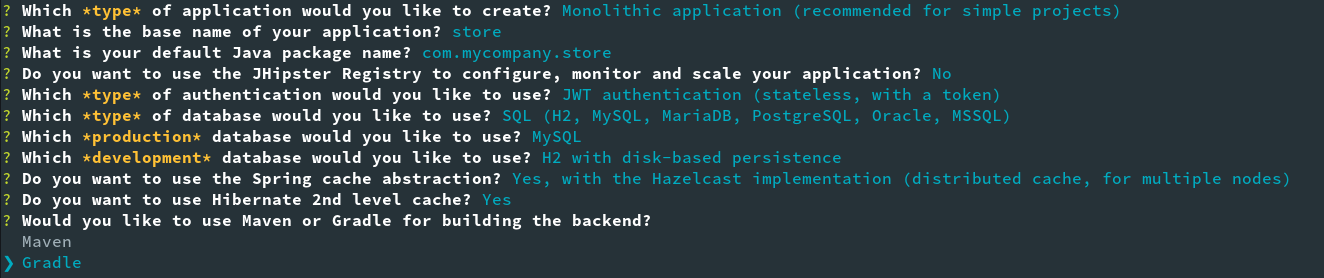
* **Question 8**: This prompt asks us to choose a Spring cache implementation. We have the option to choose between no cache, EHCache, Hazelcast, and Infinispan. Since we learned about these in the previous chapter, let's go ahead and choose Hazelcast here:



* **Question 9**. This prompt asks us to choose if we need a 2nd level cache for Hibernate. Let's choose Yes. It will use the same cache implementation we chose for the previous question:

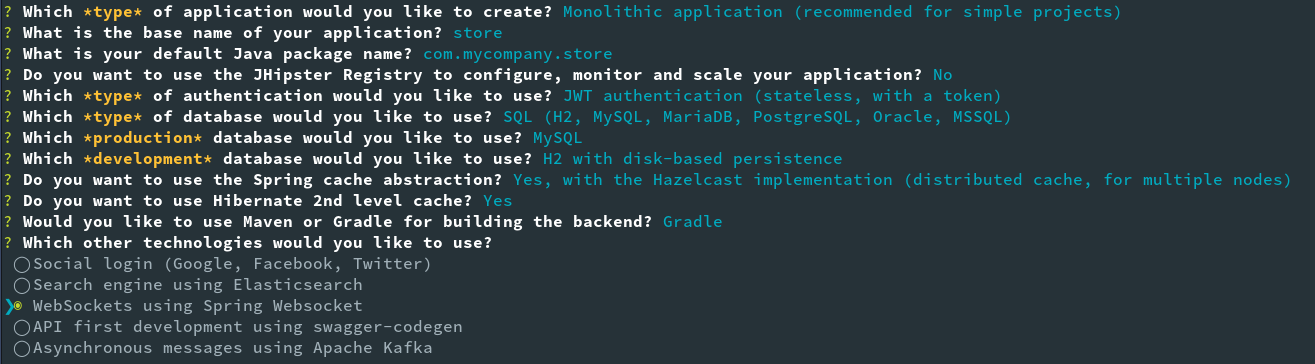


* **Question 10**: This prompt gives us the choice of the build tool to use for the project; the options are Maven and Gradle. Let's choose Gradle here as it is more modern and powerful:



* **Question 11**: This prompt is interesting as it presents various additional options supported by JHipster. The options are:
  + Social login: Adds support for using a Social login provider like Facebook, Twitter, and so on for login(Social login option is removed in JHipster 5 and you need to choose OAuth 2.0/OIDC Authentication instead to use Social login provided by the OIDC provider)
  + Elasticsearch: Adds Elasticsearch support for the generated entities
  + WebSockets: Adds WebSocket support using Spring WebSocket, SocketJS, and Stomp protocol
  + API first development with swagger-codegen: Adds Swagger codegen support for API first development
  + Apache Kafka: Adds support for asynchronous queue using Kafka

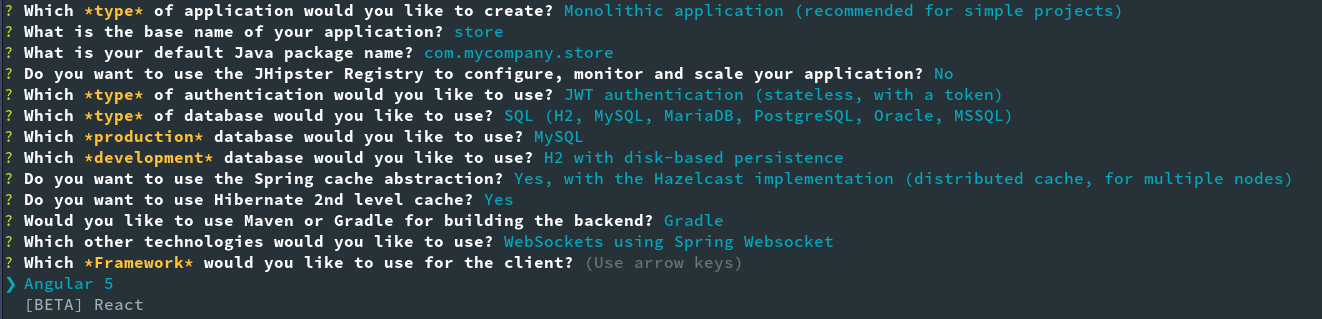
Let's keep it simple and choose WebSockets using Spring WebSocket:



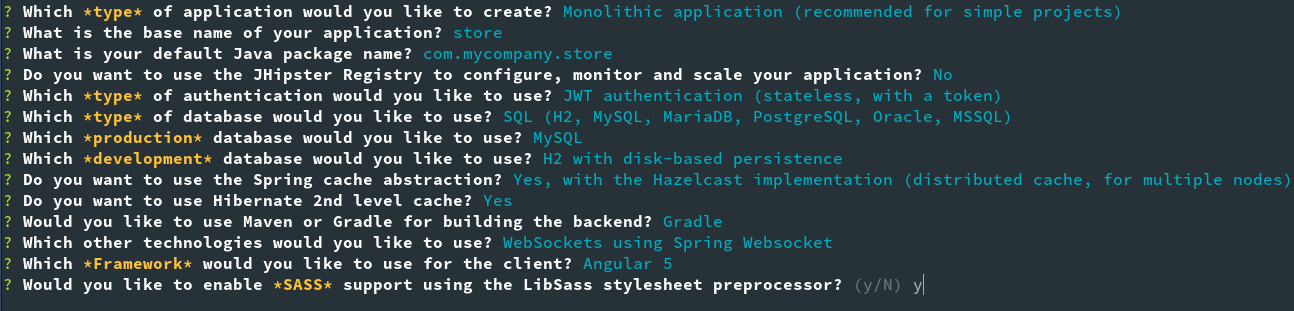
**Client-side options**

Now, the generator will ask us about the client side option, 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 5 and React. Let's choose Angular 5 here:



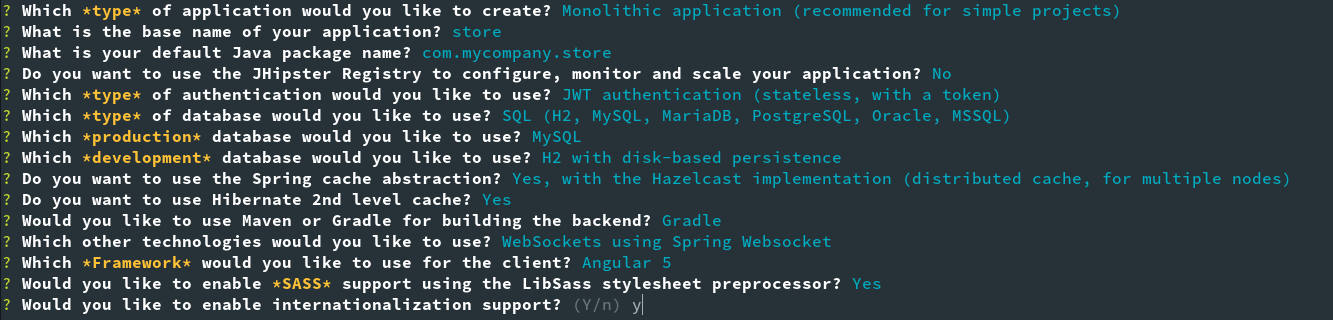
* **Question 2**. This prompt lets us enable SASS support for our CSS, and since SASS is awesome, let's enable it by selecting Yes:



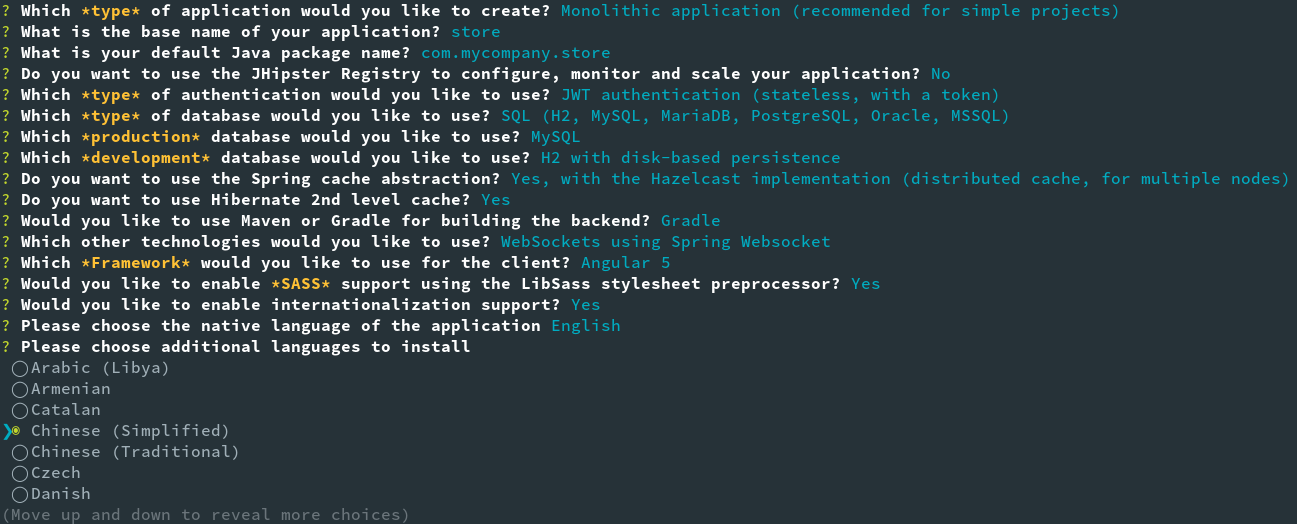
**Internationalization options**

We will now have the opportunity to enable internationalization and select the languages we would like:

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



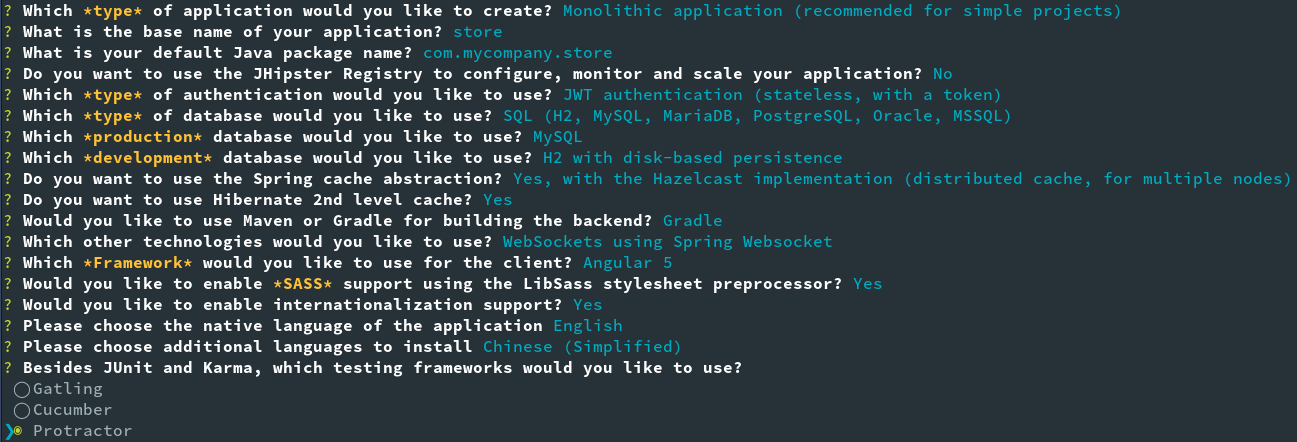
* **Question 2**: Since we enabled i18n, we will be given the option to choose a primary language and additional i18n languages. At the time of writing, there are 36 supported languages including 2 **RTL** (**Right to Left**) languages. Let's choose English as the primary language and Simplified Chinese as the additional language:



# Testing

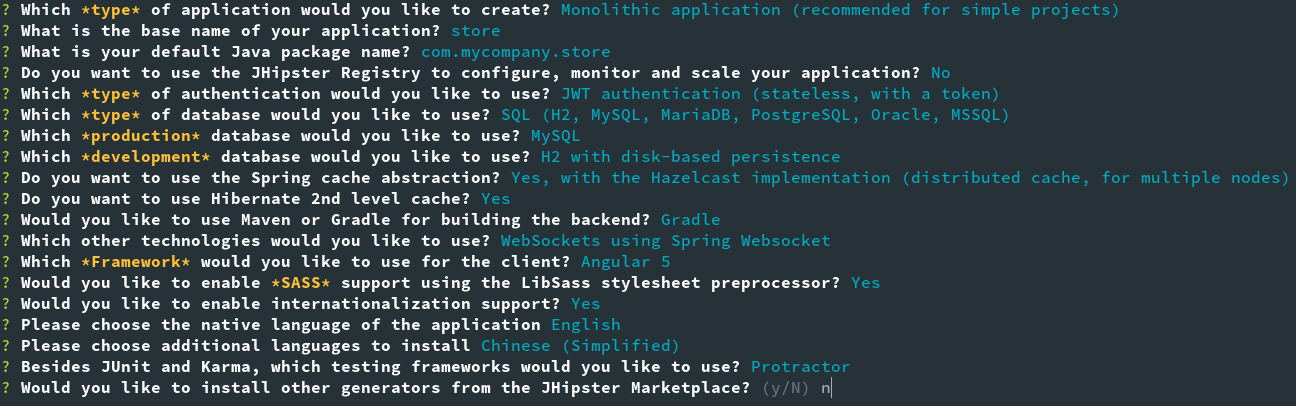
Now, we can choose testing options for our application.

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 here:



# Modules

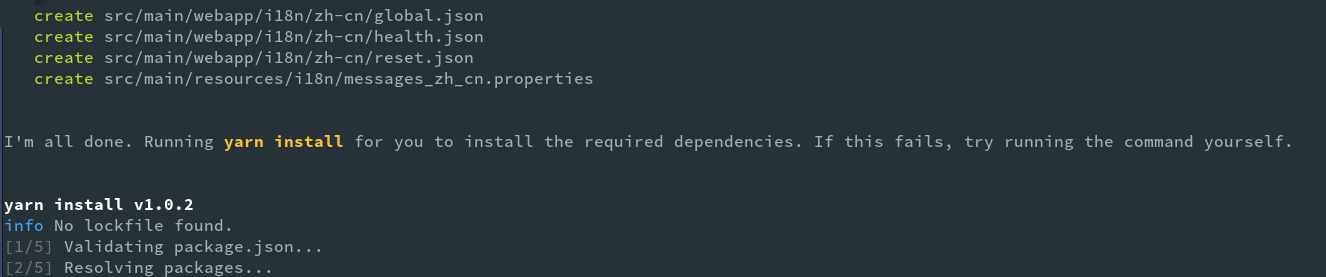
This prompt lets us choose additional third-party modules from the JHipster marketplace (<https://www.jhipster.tech/modules/marketplace>). This can be helpful if we want to use additional features not supported directly by JHipster. We will look at this in later chapters. For now, let's choose No. Don't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



Once all the questions are answered, the code generation will start and you will see an output like following, listing the files created, and then running yarn installation to get all the frontend dependencies installed.

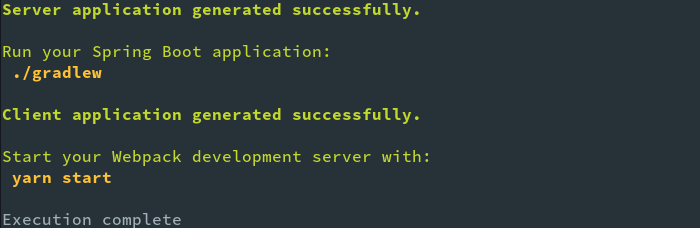
If you do not want the Yarn install and Webpack build steps to run, you could use the --skip-install flag while running JHipster to skip this. Just run jhipster --skip-install

Once the installation is complete, the generator will trigger a webpack build for the client side so that when we start the application, we have everything ready:



JHipster will check your environment to see if all the required dependencies like Java8, 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 successful messages as follows, and instructions to start the application:



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

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

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

You could also use a GUI tool like Sourcetree or GitKraken if you wish to do so to work with Git.

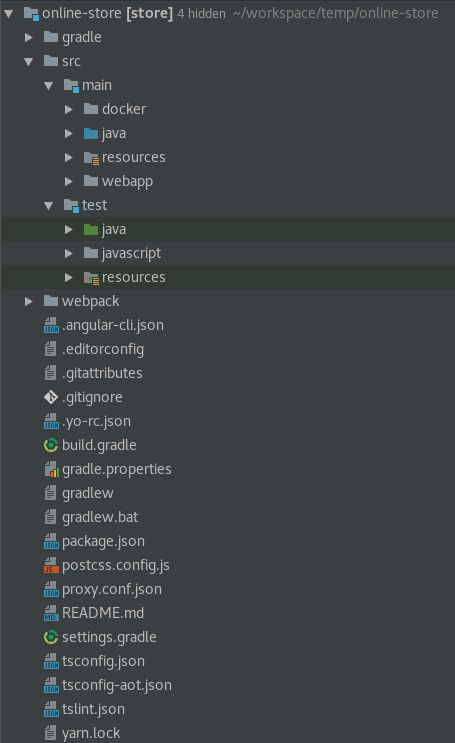
# Code walkthrough

Now that we have generated our application with JHipster, let's go through important pieces of the source code which have 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 menu option File | New | Project from existing sources and select the project folder before selecting Gradle from the options and click Next and then Finish. If you are using Eclipse, open the File | Import... dialog and select Gradle Project in the list and follow the instructions.

**File structure**

The created application will have the following file structure:



As you can see, the root folder is quite busy with a few folders but a lot of configuration files. The most interesting among them is:

* 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 has Gradle wrapper and additional Gradle build scripts which 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 which specifies our applications build lifecycle. It also has the server side dependencies specified. The build uses properties defined in the gradle.properties file alongside it. You can 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 is the NPM configuration file which 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 **AOT** (**Ahead-of-Time**) compilation.
* tslint.json: This is the lint configuration for Typescript.

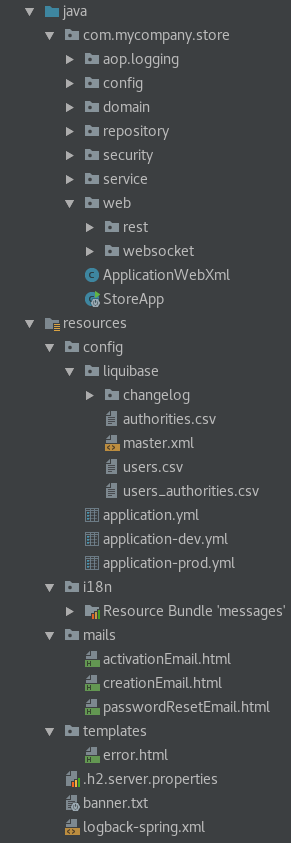
Install and configure Typescript and the Tslint plugin for your IDE or editor to make the most out of Typescript.

Now, let's take a look at the source folder. It has a main folder and a test folder, which holds the main app source code and tests the source code accordingly. The folder structure is as follows:

* main:
  + docker: Holds the Dockerfile for the application and Docker configurations for the selected options
  + java: Holds the main Java source code for the application
  + resources: Holds Spring Boot configuration files, Liquibase changelogs, and static resources like server-side i18n files and email templates used by the application
  + webapp: Holds the Angular application source code and the client side static content like 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 like server-side i18n files and email templates used by the application for test

# Server-side source code

The server-side code is situated in the Java and resources folder under src/main, as seen in the preceding screenshot. The folder structure is as follows:



You may notice that the Spring components do not use the traditional @Autowired or @Inject annotations for dependency injection in the 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. Constructor injection is a suggested best practice by the Spring team. Constructor injection also makes unit testing components easier.

**Java source**

The important parts of the Java source code are:

* 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 just 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:

**@ComponentScan**  
**@EnableAutoConfiguration**(exclude = {MetricFilterAutoConfiguration.class, MetricRepositoryAutoConfiguration.class})  
**@EnableConfigurationProperties**({LiquibaseProperties.class, ApplicationProperties.class})

* + The first one, @ComponentScan, tells the Spring application to scan the source files and auto detect Spring components (Services, Repository, Resource, Configuration classes that define Spring beans, and so on).
  + The second one is @EnableAutoConfiguration, which tells Spring Boot to try to guess and auto-configure beans that the application might need based on the classes found on the classpath and the configurations we have provided. The exclude settings specifically tells Spring Boot not to auto-configure the specified beans.
  + The third one, @EnableConfigurationProperties, helps 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) throws UnknownHostException {  
 SpringApplication app = new SpringApplication(StoreApp.class);  
 DefaultProfileUtil.addDefaultProfile(app);  
 Environment env = app.run(args).getEnvironment();  
 ...  
}

* 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 of the important ones are:
  + CacheConfiguration.java: This class configures the Hibernate second level cache for the application. Since we chose Hazelcast as the cache provider, this class configures the same way.
  + DatabaseConfiguration.java: This class configures the database for the application and enables transaction management, JPA auditing, and JPA repositories for the application. It also configures Liquibase to manage DB migrations and the H2 database for development.
  + SecurityConfiguration.java: This is a very important part of the application as it configures security for the application. Let's take a look at important parts of the class:
    - The annotations enable web security and method level security so that we can use @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 Spring security configuration:

@Override  
public void configure(WebSecurity web) throws Exception {  
 web.ignoring()  
 .antMatchers(HttpMethod.OPTIONS, "/\*\*")  
 .antMatchers("/app/\*\*/\*.{js,html}")  
 .antMatchers("/i18n/\*\*")  
 .antMatchers("/content/\*\*")  
 .antMatchers("/swagger-ui/index.html")  
 .antMatchers("/api/register")  
 .antMatchers("/api/activate")  
 .antMatchers("/api/account/reset-  
 password/init")  
 .antMatchers("/api/account/reset-  
 password/finish")  
 .antMatchers("/test/\*\*")  
 .antMatchers("/h2-console/\*\*");  
}

* + - 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):

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 http  
 ...  
 .and()  
 .authorizeRequests()  
 .antMatchers("/api/register").permitAll()  
 ...  
 .antMatchers("/api/\*\*").authenticated()  
 .antMatchers("/websocket/tracker")  
 .hasAuthority(AuthoritiesConstants.ADMIN)  
 .antMatchers("/websocket/\*\*").permitAll()  
 .antMatchers("/management/health").permitAll()  
 .antMatchers("/management/\*\*")  
 .hasAuthority(AuthoritiesConstants.ADMIN)  
 .antMatchers("/v2/api-docs/\*\*").permitAll()  
 .antMatchers("/swagger-resources/configuration/ui").permitAll()  
 .antMatchers("/swagger-ui/index.html")  
 .hasAuthority(AuthoritiesConstants.ADMIN)  
 .and()  
 .apply(securityConfigurerAdapter());  
}

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

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/>).
* It 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.
* It is disabled by default in prod mode for both monoliths and microservices, for security reasons.
* domain: The domain model classes for the application are in this package. These are simple POJOs which have JPA annotations mapping it to a Hibernate entity. When the Elasticsearch option is selected, these also act as the Document object. 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 used at 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 the objects into JSON which are to be returned in the REST API requests. This is especially useful with Hibernate as it avoids circular references between relationships, which create tons of SQL DB requests and fail:

@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)  
private String password;

* + The relationships between the database tables are also mapped to the entities using JPA annotations. Here, 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 the Spring Data repositories for the entities. These typically interface definitions which 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> findAllByActivatedIsFalseAndCreatedDateBefore(Instant   
 dateTime);  
  
 Optional<User> findOneByResetKey(String resetKey);  
  
 Optional<User> findOneByEmailIgnoreCase(String email);  
 ...  
}

* + The  @Repository annotation marks this as a Spring data repository component.
  + The interface extends JpaRepository, which lets it inherit all the default CRUD operations like findOne, findAll, save, count, and delete.
  + Custom methods are written as simple method definitions following the 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 utils, and 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 consisting of Spring service beans, DTOs, Mapstruct DTO mappers, and service utilities.
* web: This package holds the 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:
    - The 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 below 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 @Timed annotation is used to measure the performance of the method. The @Secured annotation restricts the method access to the specified role:

**@PostMapping("/users")**  
**@Timed**  
**@Secured**(AuthoritiesConstants.ADMIN)  
public ResponseEntity createUser(@Valid @RequestBody ManagedUserVM managedUserVM) throws URISyntaxException {  
 ...  
}

* + WebSocket: This package holds the Websocket controllers and view models.

JHipster uses **DTO**(**Data Transfer Object**) and **VM** (**View Model**) on the server side. DTOs are for transferring data from the service layer to and from the resource layer. They **break** the Hibernate transactions and avoids further lazy loading from being triggered by the resource layer. VMs are only used for displaying data on the web frontend and don't interact with the service layer.

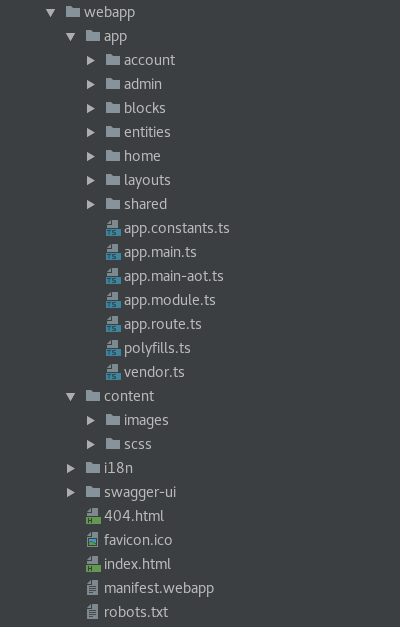
**Resources**

The important parts of resources are:

* config: This holds the application property YAML files and Liquibase changelogs. The application.yml file holds configurable Spring Boot, JHipster, and application-specific properties while the 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 the server-side i18n resource files.
* mails: This holds Thymeleaf templates for emails.
* templates: This holds Thymeleaf templates for the client side.

**client-side source code**

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



The most noteworthy among these are:

* 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 **JIT** (**Just-in-time**) compilation in the browser. This is ideal for development:

platformBrowserDynamic().bootstrapModule(StoreAppModule)  
.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 providers, and imports other modules for the application. It also bootstraps the main application component:

@NgModule({  
 imports: [  
 BrowserModule,  
 ...  
 StoreEntityModule,  
 // jhipster-needle-angular-add-module JHipster   
 will add new module here  
 ],  
 declarations: [  
 JhiMainComponent,  
 ...  
 FooterComponent  
 ],  
 providers: [  
 ProfileService,  
 ...  
 UserRouteAccessService  
 ],  
 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.
  + blocks: This folder consists of HTTP interceptors and other configs used by the application.
  + entities: This is where entity modules will be created.
  + home: The homepage module.
  + layouts: This folder has layout components like the navbar, footer, error pages, and so on.
  + shared: This module contains all the shared services (auth, tracker, user), components (login, alert), entity models, and utilities required for the application.
* content: This folder contains static content like images, CSS, and SASS files.
* i18n: This is where the i18n JSON files live. Each language has a folder with numerous JSON files organized by modules.
* swagger-ui: This folder has the Swagger UI client used in development for 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 some commented out utility code like 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 the sw.js.

**Starting the application**

Now, let's start the application and see 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

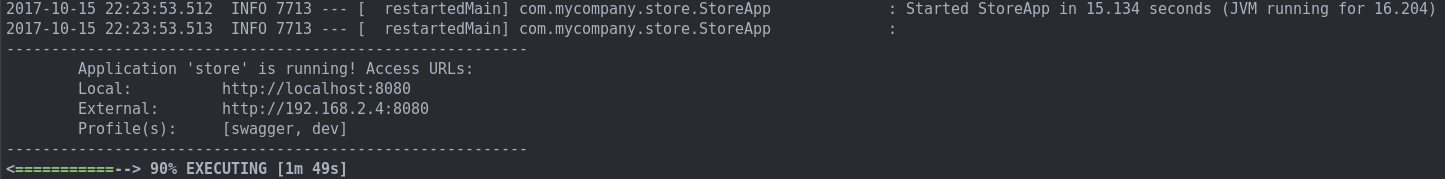
Let's start the application using the Gradle task. If you want to run the application directly in the IDE, just open the main app file StoreApp.java mentioned earlier, right-click, and choose Run 'StoreApp'.

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 bootRun:

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

Running ./gradlew is equivalent to running ./gradlew bootRun -Pdev. For the client side, the webpack build needs to be run before starting the server for the first time, otherwise you will see a blank page. This task is run automatically during the app generation, but if it fails for some reason, it can be triggered manually by running yarn run webpack:build. The task can be triggered directly by the Gradle command as well by running ./gradlew webpackBuildDev bootRun -Pdev.

Gradle will start downloading the wrapper and dependencies, and you should see the console output similar to the following screenshot after some time (anywhere from a few seconds to a few minutes depending on network speed):



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

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

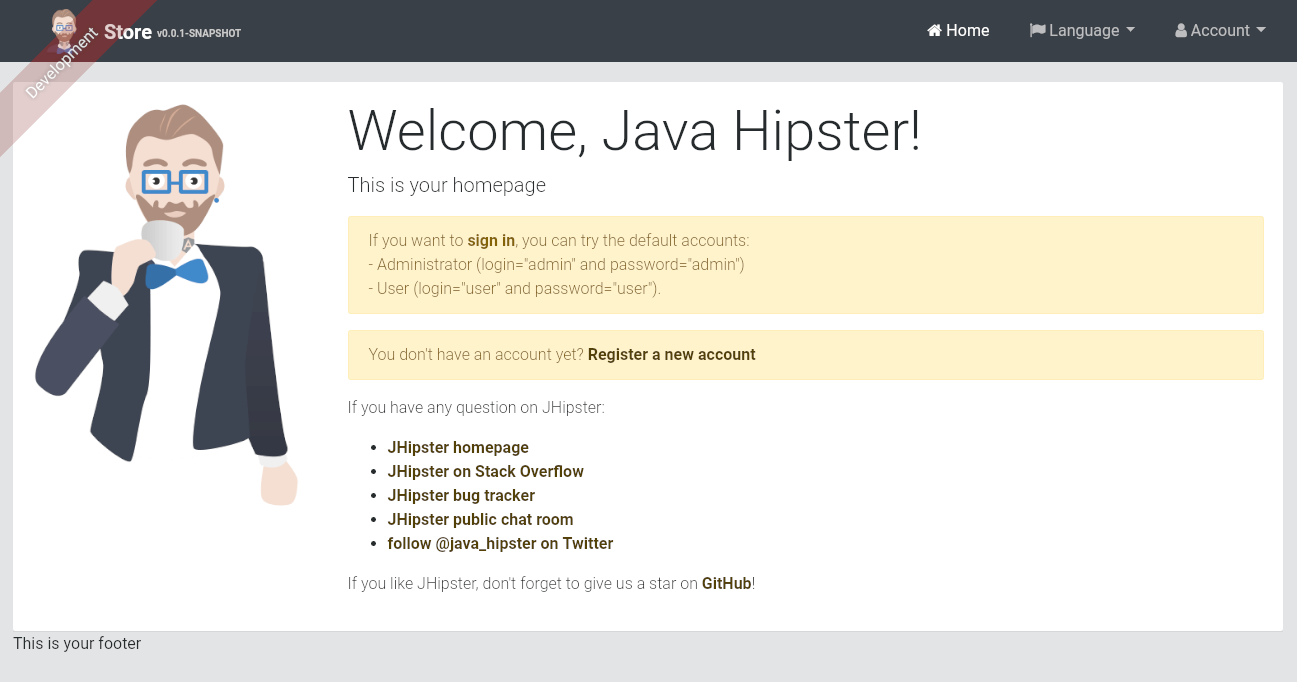
**Application modules**

Let's see the different modules available out of the box. The modules can be grouped into:

* Home and Login
* Account
* Admin

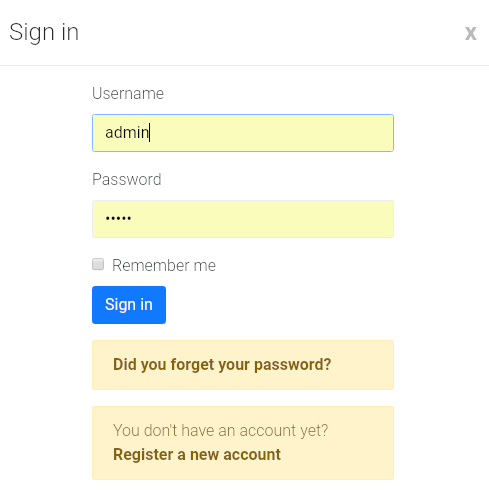
**Home and Login modules**

Once you open the URL, you will see a cool-looking hipster drinking coffee on the homepage as follows:

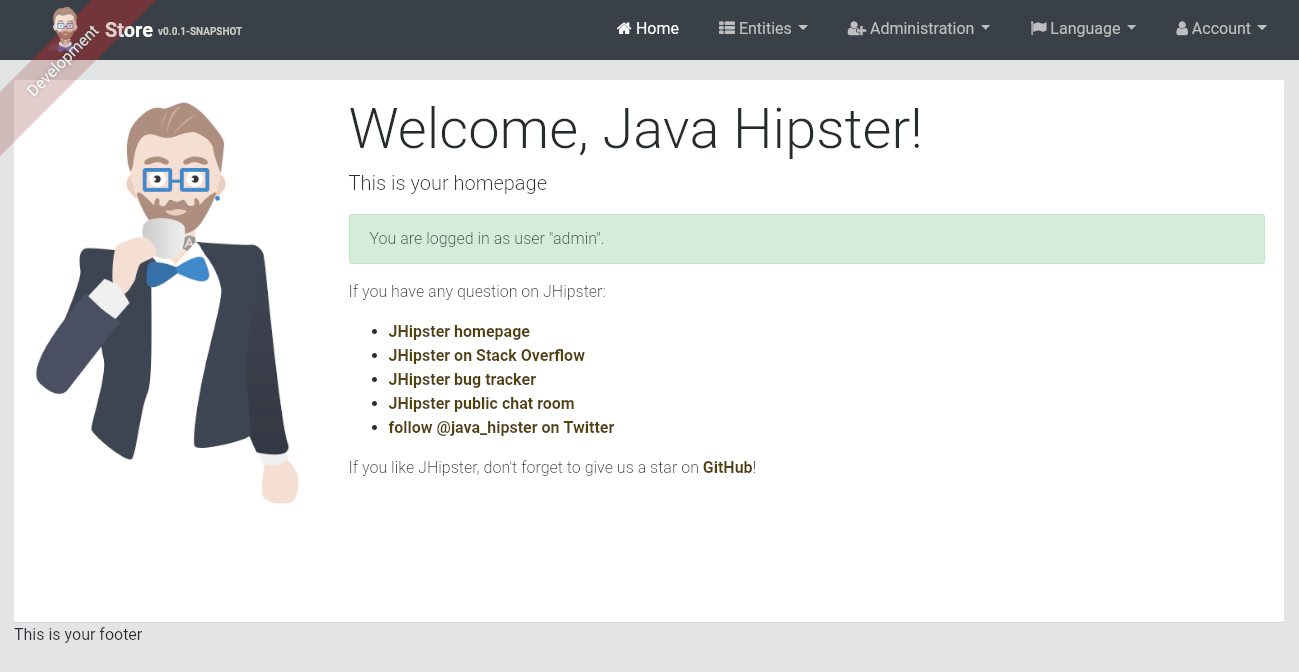


This is the home page. Let's log in to the application using the default credentials.

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—Username—admin, Password—admin, and click *Sign in*:



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



1. 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:



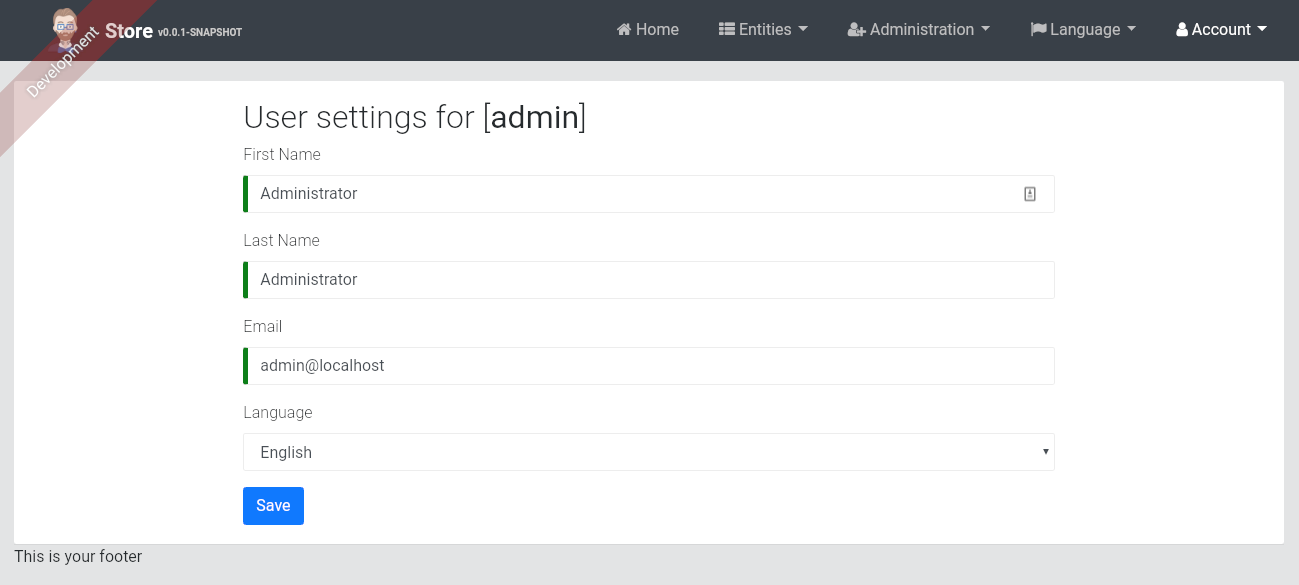
**Account modules**

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

* Settings
* Password
* Registration

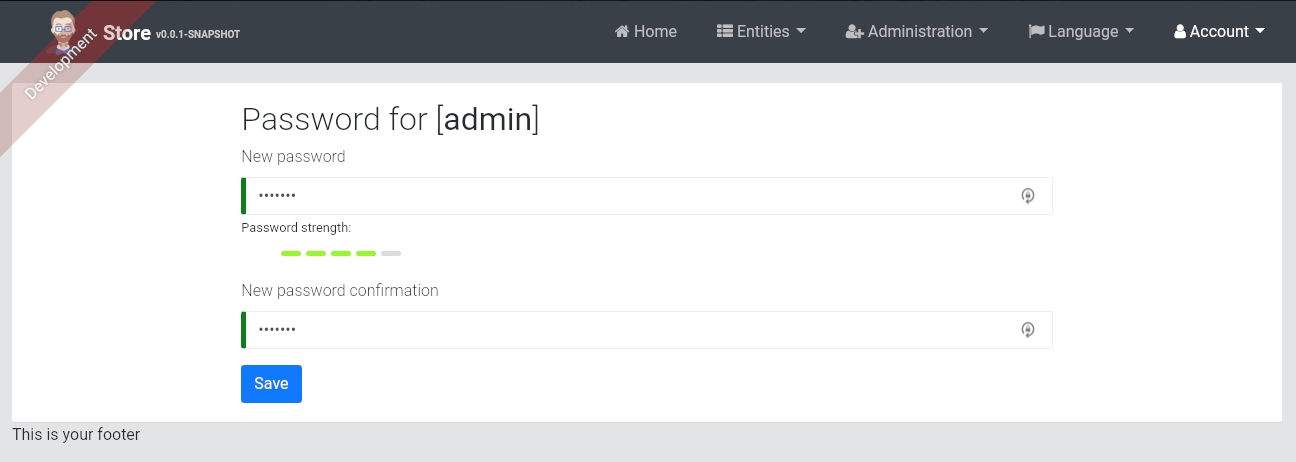
# Settings

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



# Password

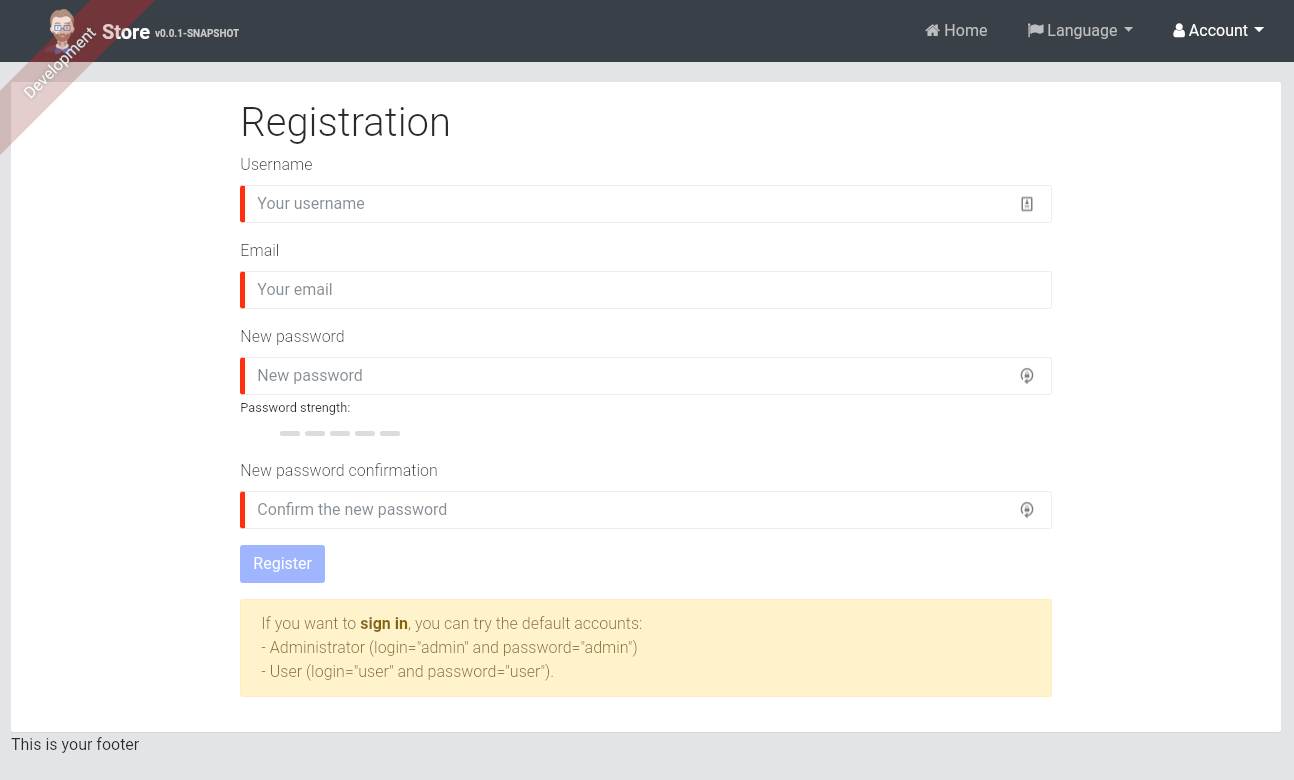
This module lets you change the password for the current user. There is also a forgot password flow with email verification out of the box:



To use the email features, you will have to configure an SMTP server in the application properties. We will look at this in a later chapter.

# Registration

This module is available only when you are not logged in. This lets you signup/register as a new user for the application. This will trigger a user activation flow with an activation email and verification. This module will not be available when choosing Oauth2 as your authentication:



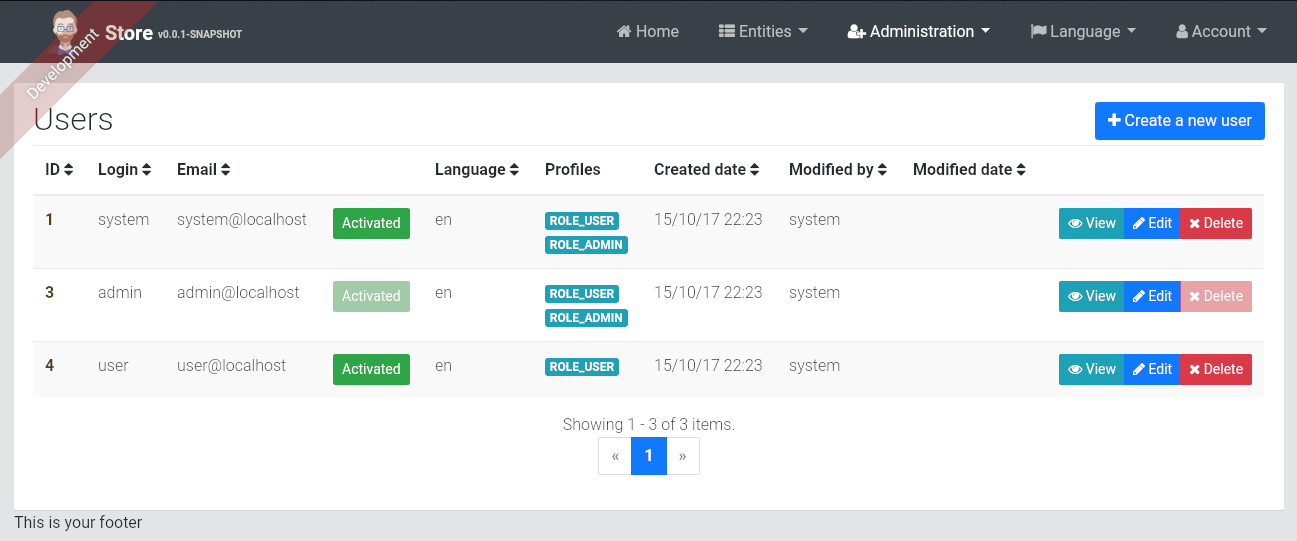
**Admin module**

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

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

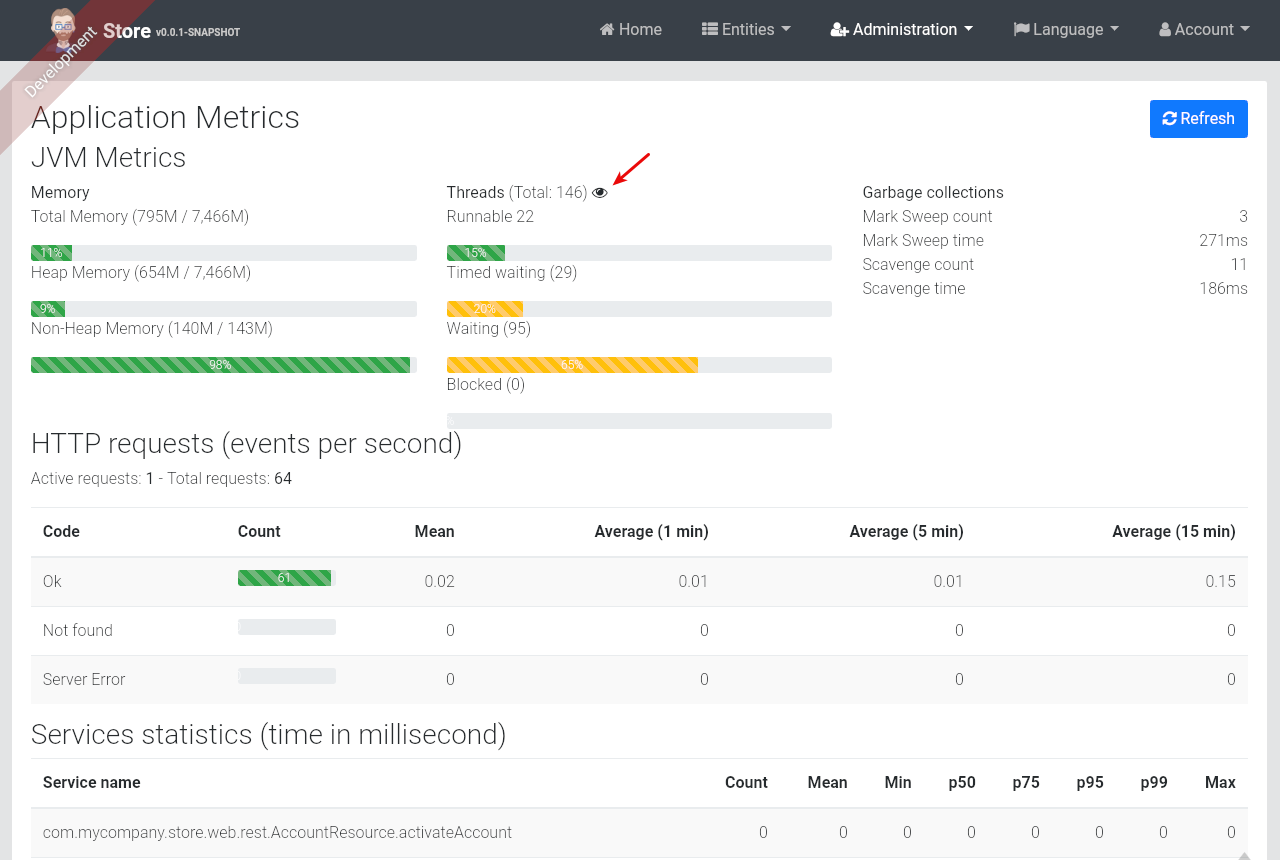
# User management

This module provides you with CRUD functionality to 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:



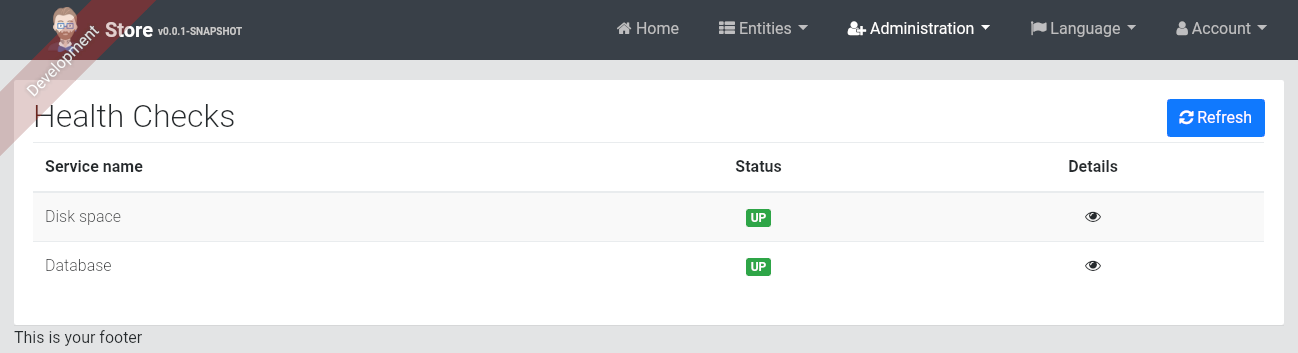
# Metrics

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 eye icon near Threads will let you see the thread dump as well:



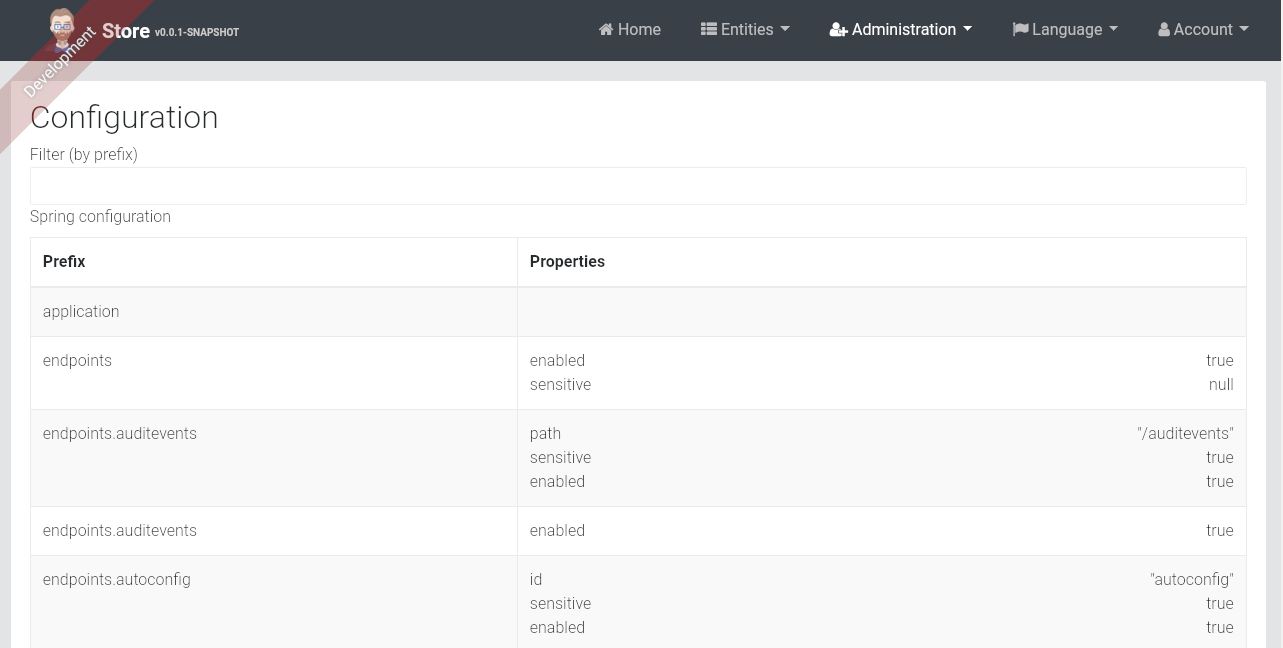
# Health

This module provides the health status of application components like Database and other info like Disk space:



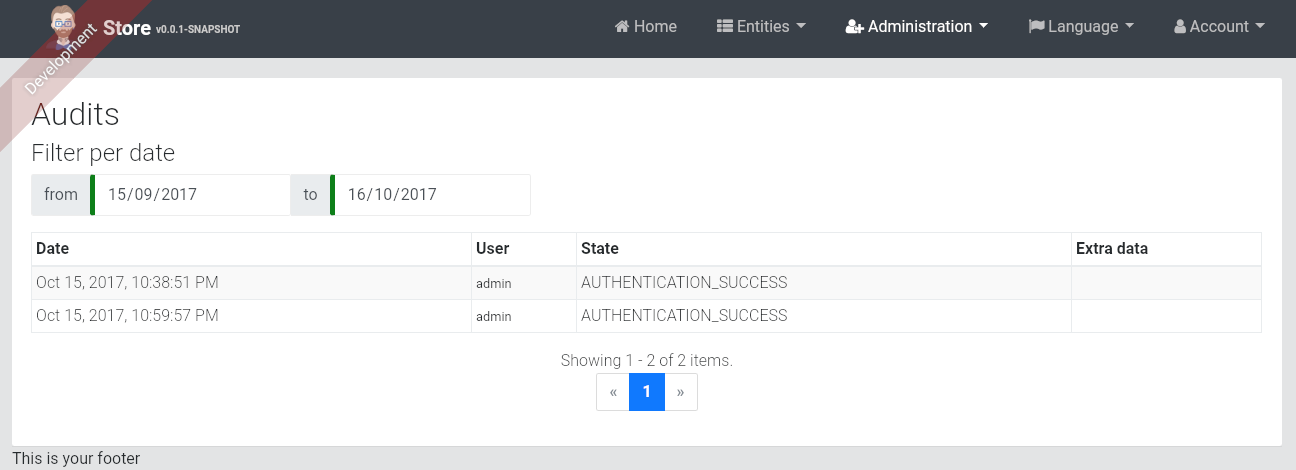
# Configuration

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



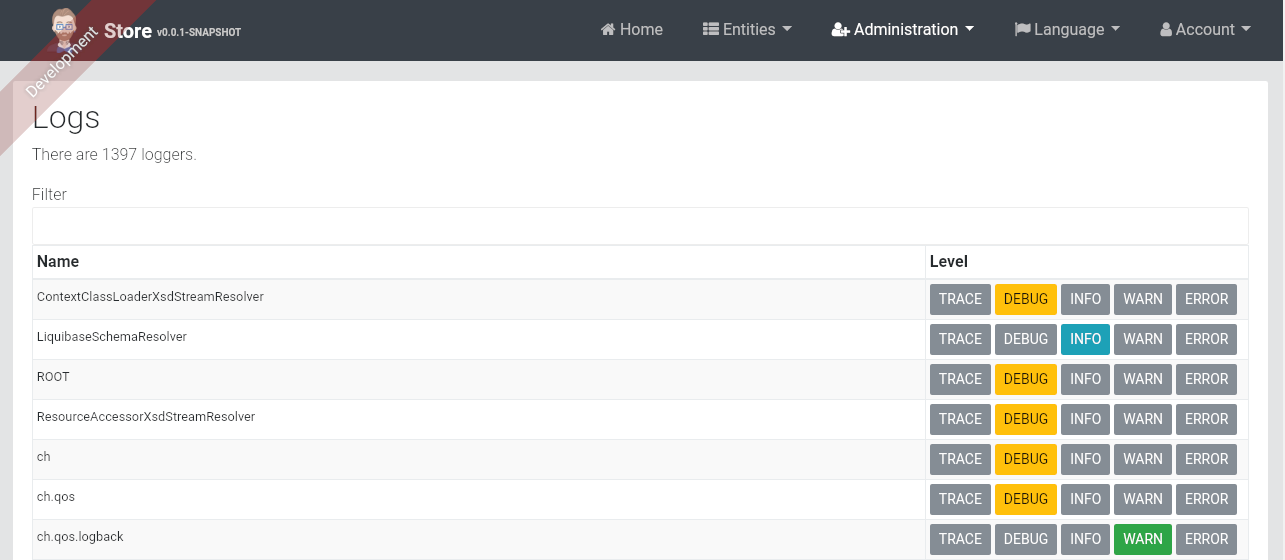
# Audits

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



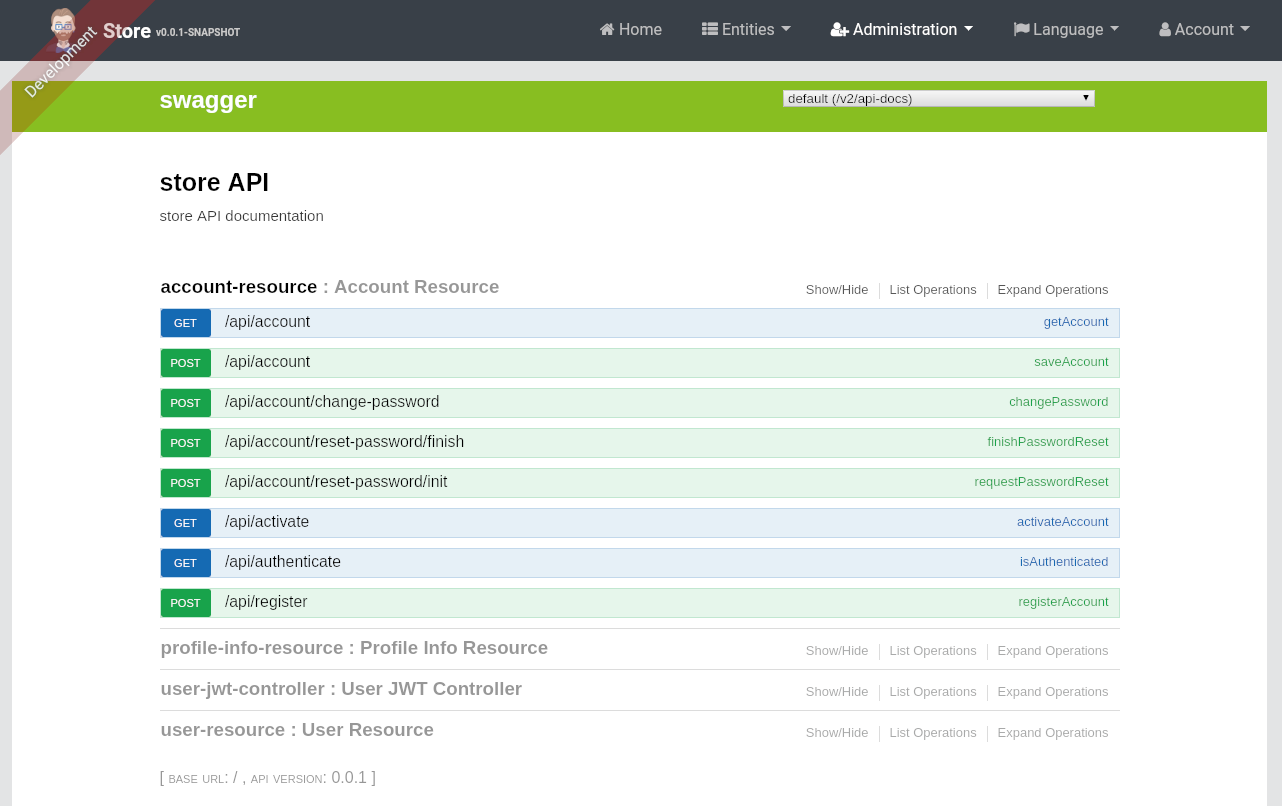
# Logs

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



# API

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



# 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 to choose even more. Let's run the 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.

# Server-side tests

The 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 a Gradle test task. 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 shown here:

**> ./gradlew test  
...  
BUILD SUCCESSFUL in 45s  
8 actionable tasks: 6 executed, 2 up-to-date**

# Client-side tests

The client-side unit tests and end-to-end tests are available under src/test/javascript.

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

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

Let's run them using the npm scripts. First, let's run the Karma unit tests. In the terminal, execute the following code. You can also use npm instead of yarn if you prefer:

**> yarn test**

It should produce a similar output in the end:

**PhantomJS 2.1.1 (Linux 0.0.0): Executed 56 of 56 SUCCESS (1.672 secs / 1.528 secs)**  
**=============================== Coverage summary ===============================**  
**Statements : 69.25% ( 903/1304 )**  
**Branches : 40.43% ( 112/277 )**  
**Functions : 48.89% ( 154/315 )**  
**Lines : 67.72% ( 795/1174 )**  
**================================================================================**  
**Done in 37.25s.**

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

**> yarn e2e**

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

**Started**  
**..........**  
**10 specs, 0 failures**  
**Finished in 11.776 seconds**  
  
**[00:02:57] I/launcher - 0 instance(s) of WebDriver still running**  
**[00:02:57] I/launcher - chrome #01 passed**

# Summary

In this chapter, we saw how to create a monolithic web application using JHipster. We also walked through important aspects of the created source code and learned how to run the created application and the automated tests. We also browsed through the created modules and saw them in action. In the next chapter, we will see how we can utilize JHipster to model our business use case and generate entities for them. We will also learn about the **JHipster Domain Language** (**JDL**).

**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.

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 DSL to describe the entities and their relationships (for SQL databases only).

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 .jh extension.

Visit <http://www.jhipster.tech/jdl/> for 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.

**DSL grammar for JDL**

Now, let's see the JDL grammar. At the time of writing, JDL supports generating complete entity models with relationships and options such as DTO, service layer, and so on. The grammar can be broken down into the following:

* Entity declaration
* Relationship declaration
* Options declaration

In the following syntax, [] denotes optional and \* denotes 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 // syntax.

It is also possible to define numerical constants in JDL, for example, DEFAULT\_MIN\_LENGTH = 1.

# Entity modeling with JDL

The entity declaration is done using the following syntax:

entity <entity name> ([<table name>]) {  
 <field name> <type> [<validation>\*]  
}

<entity name> 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.

<field name> is the name of the fields (attributes) you want for the entity and <type> is the field type, as in String, Integer, and so on. Refer to <http://www.jhipster.tech/jdl/#available-types-and-constraints> for all supported field types. The ID field will be automatically created and hence need not be specified in JDL.

<validation> is optional and one or more <validation> 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.

An example entity declaration would look like the following:

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

Enumerations can also be declared using the following syntax:

enum <enum name> {  
 <VALUE>\*  
}

Here is an example:

enum Language {  
 ENGLISH, DUTCH, FRENCH  
}

# Relationship management

The relationship between entities can be declared using this syntax:

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

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

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

<relationship name> is optional and can be used to specify the field names to create for the relationship in the domain object. <display field> 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, by default the ID field will be used. <validation> can be specified on the <from entity> or <to entity> and is optional. Currently, only required is supported.

OneToMany and ManyToMany relationships are always bidirectional in JHipster. In case of ManyToOne and OneToOne relationships, it is possible to create both bidirectional and unidirectional relationships. For unidirectional relationships, just skip the <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 relations are also possible with a user entity.

**DTO, service, and pagination options**

JDL also allows us to declare entity related options easily. Options currently supported are:

* 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 andserviceImpl. 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 created, which may not be desirable in some situations and you might want to use an intermediatory **Data Transfer Object** (**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 to use a service layer when using DTO. A possible value is mapstruct. For more info 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 pager, pagination, and infinite-scroll.
* noFluentMethod: This lets us disable Fluent API style setters for the generated entity domain objects.
* skipClient/skipServer: These options let us either skip the client-side code or server-side code during generation.
* angularSuffix: This option lets us specify a suffix for the folder and class names in the frontend code.

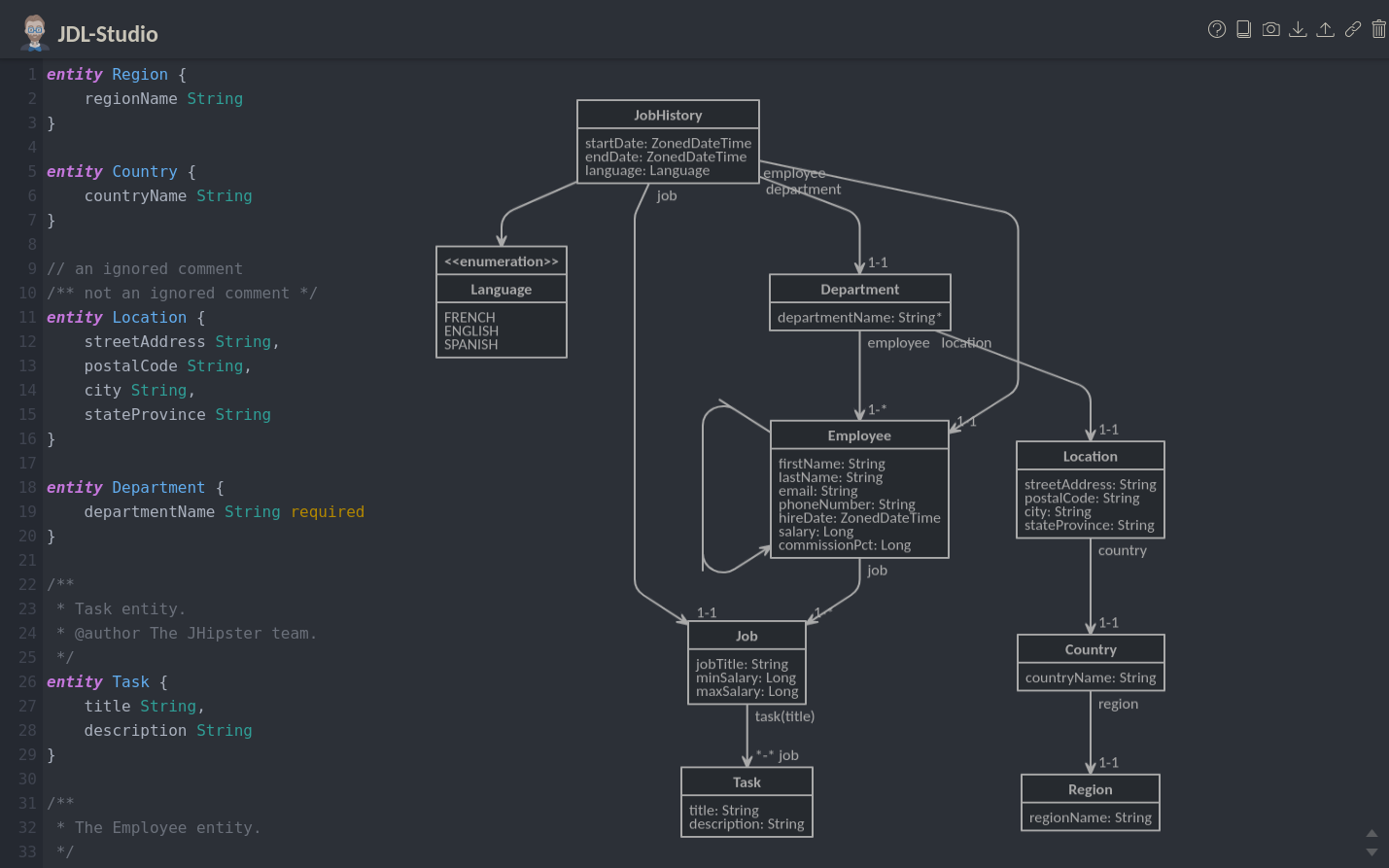
The general syntax for option declaration is <OPTION> <ENTITIES | \* | all> [with <VALUE>] [except <ENTITIES>].

The following are some possible options and different syntax in which they can be declared:

entity A  
entity B  
...  
entity Z  
  
dto \* with mapstruct  
service A with serviceImpl  
service B with serviceClass  
paginate \* with pagination except B, C  
paginate B, C with infinite-scroll  
filter A, B

# 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:



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

Navigate your favorite browser 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.

# Use case entity model with explanation

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 name, description, price, size, and image which is a Blob. name, price, and size are required fields. price also has a min value validation. The size field is an enum with defined values.

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

1. 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, 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 detail soon.

1. 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. ProductOrder holds the placedDate and 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 min value validation. OrderStatus and OrderItemStatus are enum fields.

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

1. Add this 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

It means the OrderItem knows their Product but Product does not know about 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 made for a product we could make this bi-directional.

1. Add the following snippet for 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 both the source entity and destination entity know about each other.

We declare that a Customer can have multiple ProductOrders, ProductOrder can have multiple OrderItems and Invoices, Invoice can have many Shipment, and ProductCategory can have many Products. From the destination entity, the source entities are mapped as ManyToOne.

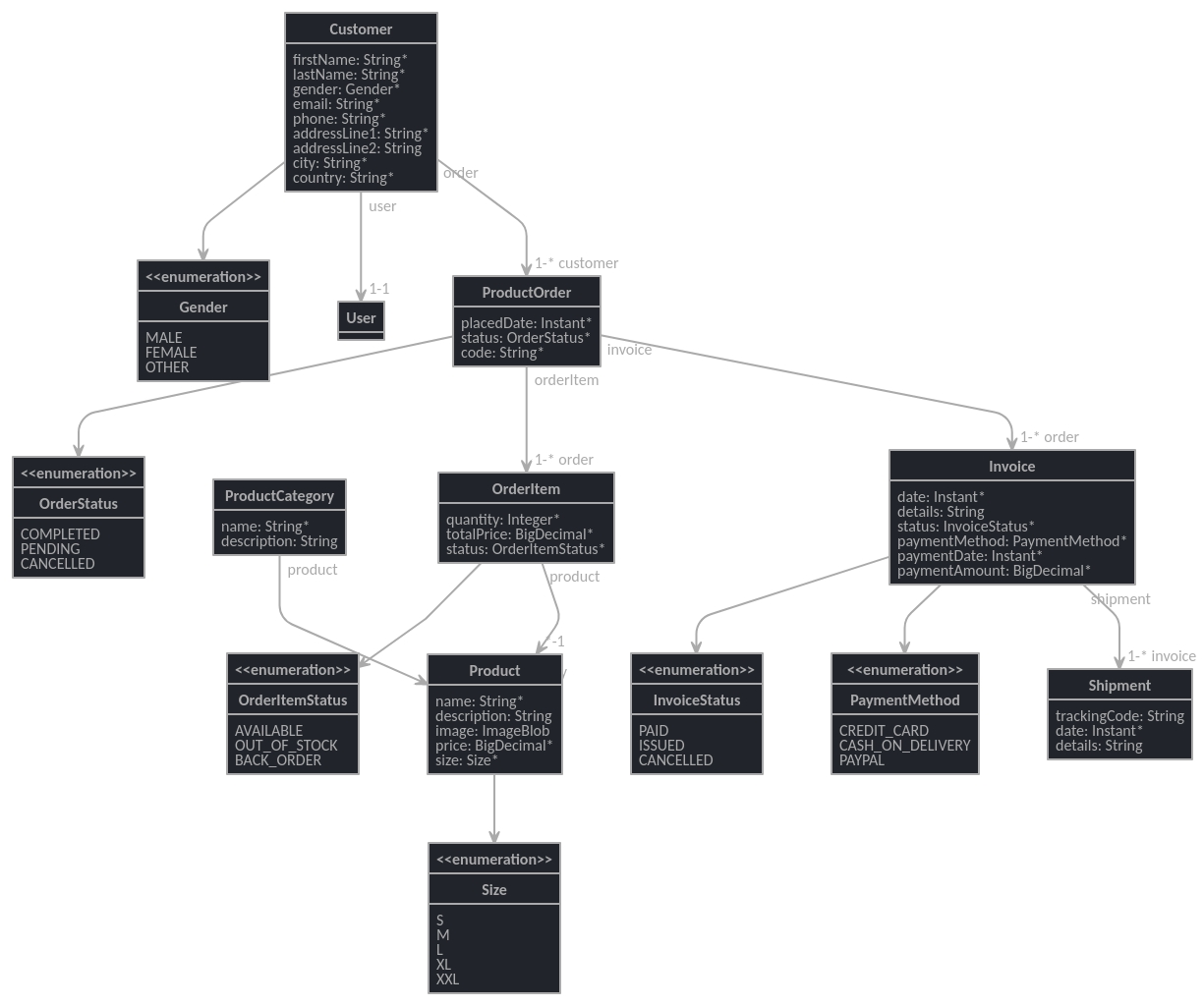
**Options for entities**

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 enabled pagination for some of the entities that may get a lot of entries over time.

The 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-hand corner of the JDL Studio application.
2. Save the file with the name online-store.jh inside the online-store directory where we created our application in the previous chapter.

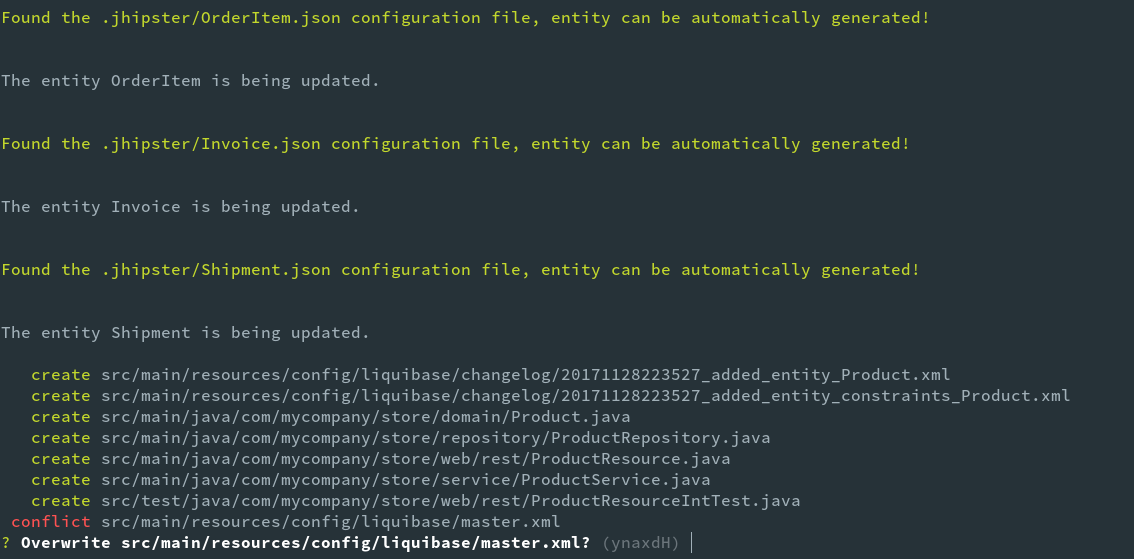
# Entity generation with JHipster

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

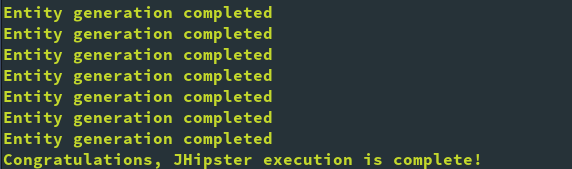
Open your favorite Terminal application and navigate to the online-store folder where we created the application earlier. Then, execute the import-jdl command:

**> cd online-store  
> jhipster import-jdl online-store.jh**

This will trigger the entity creation process and you will be asked to confirm the overwriting of existing files with changes. Take a look at the following screenshot:



Enter a to confirm the overwrite of all files with changes. Once the files are generated, JHipster will trigger a yarn webpack:build step to rebuild the client side code. Once done you will see a success message like the following:



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

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

# 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. Let's take a 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 into 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:

{   
 "fluentMethods": true,   
 "relationships": [   
 {   
 "relationshipType": "many-to-one",   
 "relationshipName": "productCategory",   
 "otherEntityName": "productCategory",   
 "otherEntityField": "id"   
 }   
 ],   
 "fields": [   
 {   
 "fieldName": "name",   
 "fieldType": "String",   
 "fieldValidateRules": [   
 "required"   
 ]   
 },   
 {   
 "fieldName": "description",   
 "fieldType": "String"   
 },   
 {   
 "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"   
 }   
 ],   
 "changelogDate": "20180114123458",   
 "javadoc": "Product sold by the Online store",   
 "entityTableName": "product",   
 "dto": "no",   
 "pagination": "pagination",   
 "service": "serviceClass",   
 "jpaMetamodelFiltering": false   
}

# Server-side source code

Now let's look at the server-side code 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;  
  
 @Lob  
 @Column(name = "image")  
 private byte[] image;  
  
 @Column(name = "image\_content\_type")  
 private String imageContentType;  
  
 @NotNull  
 @DecimalMin(value = "0")  
 @Column(name = "price", precision=10, scale=2, nullable = false)  
 private BigDecimal price;  
  
 @NotNull  
 @Enumerated(EnumType.STRING)  
 @Column(name = "jhi\_size", nullable = false)  
 private Size size;  
  
 @ManyToOne  
 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.

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

This annotation is used by Swagger to show useful documentation when the entity is used in an endpoint:

@Entity  
@Table(name = "product")

These are JPA annotations declaring the POJO as an entity and mapping it to an SQL table:

@Cache(usage = CacheConcurrencyStrategy.NONSTRICT\_READ\_WRITE)

This is a Hibernate annotation, which lets us enable level 2 cache for this entity. In our case using Hazelcast:

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

The id field is special and is mapped as a generated value field. Depending on the 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:

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

This 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:

@NotNull  
@DecimalMin(value = "0")

These are Bean validation annotations enabling validation for the fields:

@Lob  
@Column(name = "image")  
private byte[] image;  
  
@Column(name = "image\_content\_type")  
private String imageContentType;

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:

@Enumerated(EnumType.STRING)

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

@ManyToOne  
private ProductCategory productCategory;

The relationships are mapped using annotations such as @ManyToOne, @OneToMany, @OneToOne, and @ManyToMany.

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

@OneToMany(mappedBy = "productCategory")  
@JsonIgnore  
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT\_READ\_WRITE)  
private Set<Product> products = new HashSet<>();

As you can see, the relationship also specifies a cache for it. It tells Jackson to ignore the field while converting to JSON to avoid a circular reference since ProductCategory is already mapped in Product entity:

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

This is a fluent setter generated by default along with the standard setter. This can be turned off by specifying the noFluentMethod for the entity in JDL. Fluent methods are handy as they let us chain setters as follows for more concise code:

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 file names <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 do all CRUD actions using this simple interface declaration. Additional repository methods can be added here easily. We will see about that 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")  
 @Timed  
 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(page,   
 "/api/products");  
 return new ResponseEntity<>(page.getContent(), headers,   
 HttpStatus.OK);  
 }

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

# 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 product.model.ts. This maps directly to the domain object:

export class Product implements IProduct {  
 constructor(  
 public id?: number,  
 public name?: string,  
 public description?: string,  
 public imageContentType?: string,  
 public image?: any,  
 public price?: number,  
 public size?: Size,  
 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 also 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 created in product.service.ts:

@Injectable()  
export class ProductService {  
  
 private resourceUrl = SERVER\_API\_URL + 'api/products';  
  
 constructor(private http: HttpClient) { }  
  
 ...  
  
 query(req?: any): Observable<HttpResponse<Product[]>> {  
 const options = createRequestOption(req);  
 return this.http.get<Product[]>(  
 this.resourceUrl,   
 { params: options, observe: 'response' }  
 )  
 .map((res: HttpResponse<Product[]>) => this.convertArrayResponse(res));  
 }  
  
 ...  
}

As you can see, the service has a constructor with dependencies 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.

There is also ProductPopupService defined in product-popup.service.ts, a utility service to open popup dialogs for entity editing and deletion.

# Angular components of the entity

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

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 to do other actions such as alerts and event broadcasts:

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

product-dialog.component.ts defines ProductDialogComponent and ProductPopupComponent, which handle the create/edit dialog page using template product-dialog.component.html:

@Component({  
 selector: 'jhi-product-dialog',  
 templateUrl: './product-dialog.component.html'  
})  
export class ProductDialogComponent implements OnInit {  
 ...  
}  
  
@Component({  
 selector: 'jhi-product-popup',  
 template: ''  
})  
export class ProductPopupComponent implements OnInit, OnDestroy {  
 ...  
}

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

ProductDeleteDialogComponent and ProductDeletePopupComponent defined in product-delete-dialog.component.ts manages the delete popup 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: 'product/:id',  
 component: ProductDetailComponent,  
 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. The 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. Routes having a popup, declares the outlet: 'popup' attribute.

# Angular module for the entity

Finally, 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(ENTITY\_STATES)  
 ],  
 declarations: [  
 ProductComponent,  
 ProductDetailComponent,  
 ProductDialogComponent,  
 ProductDeleteDialogComponent,  
 ProductPopupComponent,  
 ProductDeletePopupComponent,  
 ],  
 entryComponents: [  
 ProductComponent,  
 ProductDialogComponent,  
 ProductPopupComponent,  
 ProductDeleteDialogComponent,  
 ProductDeletePopupComponent,  
 ],  
 providers: [  
 ProductService,  
 ProductPopupService,  
 ProductResolvePagingParams,  
 ],  
 schemas: [CUSTOM\_ELEMENTS\_SCHEMA]  
})  
export class StoreProductModule {}

The module declares the components and registers services provided by it. The module also imports shared modules so that it can access shared services and components. The module is imported by the StoreEntityModule defined in entity.module.ts under src/main/webapp/app/entities.

# Generated pages

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

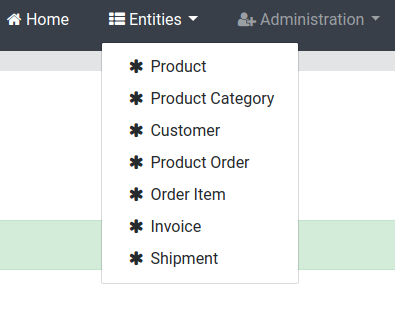
**> ./gradlew**

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 yarn start just to see the new pages, but please note that for further development it is better to use yarn start along with the preceding command. If you had the server already running while generating the entities, then no need to run this command, instead just compile the source again using the ./gradlew compileJava command. Using your IDE and Spring devtools will hot reload the application for you. If you had yarn start running then a hot reload will take place on the client side as well, otherwise, it will just refresh the page. We will see more about hot reloading in the next chapter.

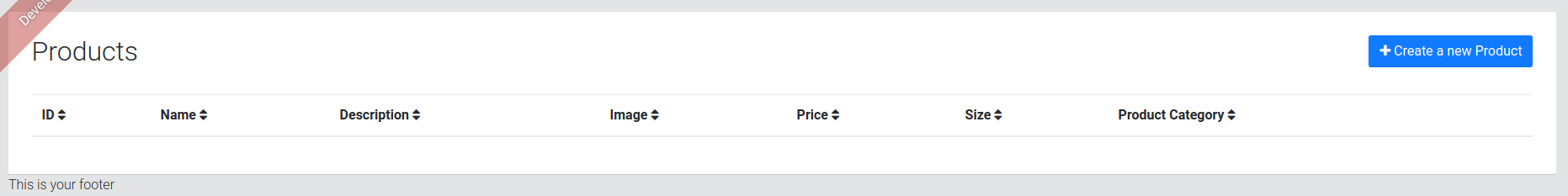
Once you see the following message, the server is ready and we can navigate to the URL http://localhost:8080 in our favorite browser:

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

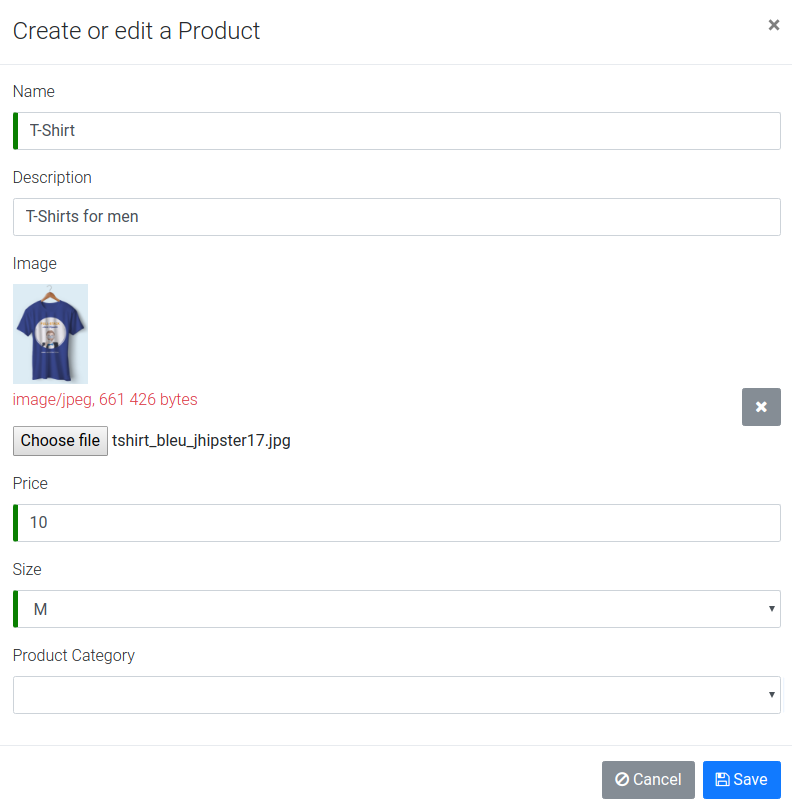
If you are not already logged in, sign in using the default admin user with the password admin by clicking on the Sign in link on the home page. Once logged in, click on the Entities link in the menu and you will see all our entities listed there:



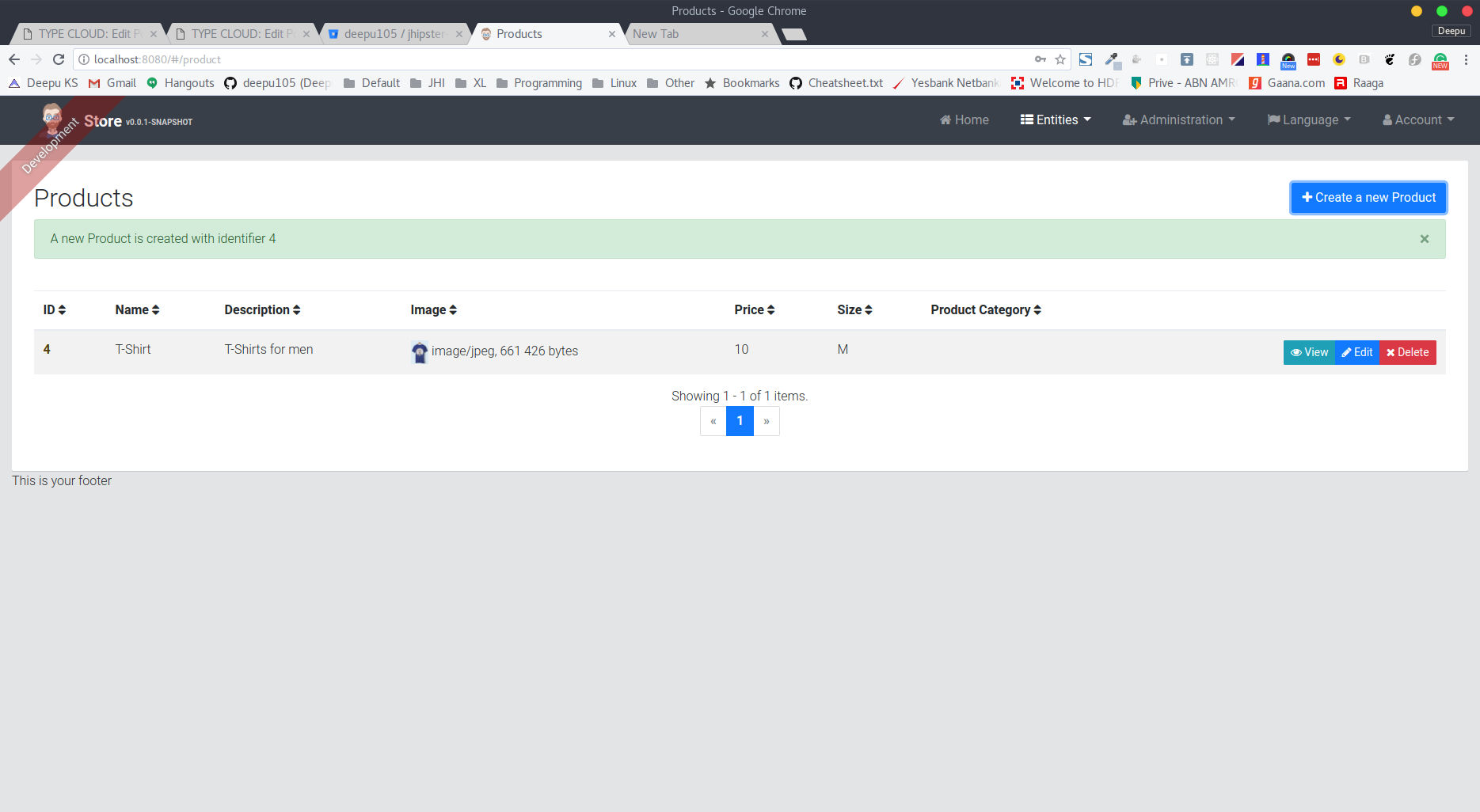
Click on the Product and you will see the Products listing screen. It doesn't have any items yet as we haven't created any:



Let's create an entity, click on the Create a new Product button on the screen and you will see the Create or edit a Product popup dialog:



Enter Name, Description, Price, and Size. Choose an image by clicking on the Choose file button. Don't worry about Product Category as we haven't created any yet. Now click on Save and the popup will disappear and the listing screen will be refreshed with the 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.

# Running generated tests

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

Let's run the server-side unit/integration tests, client-side Karma 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. Make sure you have the application running, as e2e tests will need it. If the application is not running first start it by running ./gradlew in a Terminal:

**> ./gradlew test && yarn test && yarn e2e**

# Summary

In this chapter, we saw how to model and create entities using JDL. We also walked through important aspects of the created source code. We also browsed through the created entity modules and saw them in action. In the next chapter, we will see how we can utilize JHipster to further develop the application and include specific business logic and tweaks. We will also learn about some of the technologies used in more depth.

**Customization and Further Development**

In the previous chapter, we saw how to use the JHipster Domain Language 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 application to suit our needs. We will learn about:

* Live reload with Spring DevTools and BrowserSync
* Customizing the angular frontend for an entity
* Editing an entity 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**

When developing an application, one of the most annoying and time-consuming parts is recompiling the code and restarting the servers to see the code changes 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 transpiling of client-side code, and more. So, if you are refactoring a field for an entity, you would traditionally need to do 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 them, hence affecting productivity.

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

With JHipster you get exactly that. JHipster uses Spring Boot DevTools, 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 do instant reload, but it beats doing it manually and doesn't require any extra configuration to enable it.

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

If you choose a NoSQL DB, such as MongoDB, Cassandra, or Couchbase, JHipster provides database migration tools for those 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 in 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, 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 reload on the client side you need to run yarn start, which will start the development server and open up a browser pointing to http://localhost:9000. Notice the port 9000. BrowserSync will be using this port, while the application backend will be served at 8080, which all requests will be proxied through via 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 your actions are replicated, thanks to BrowserSync. Try changing some code and save the file to see live reload in action.

# Setting up live reload 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 yarn 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.

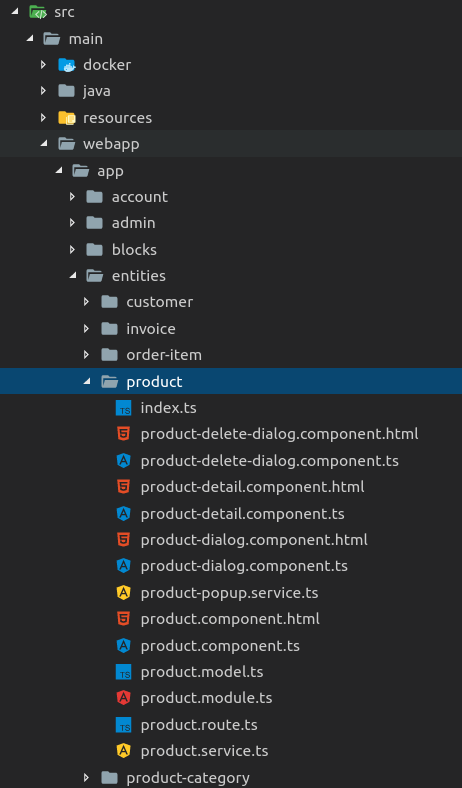
With IntelliJ IDEA, files are automatically saved and so you can set up Ctrl + S to compile the classes giving you a nice workflow. In Eclipse, saving a class automatically compiles it.

When you make changes on the client side, simply save the file and webpack dev server and BrowserSync will do the rest.

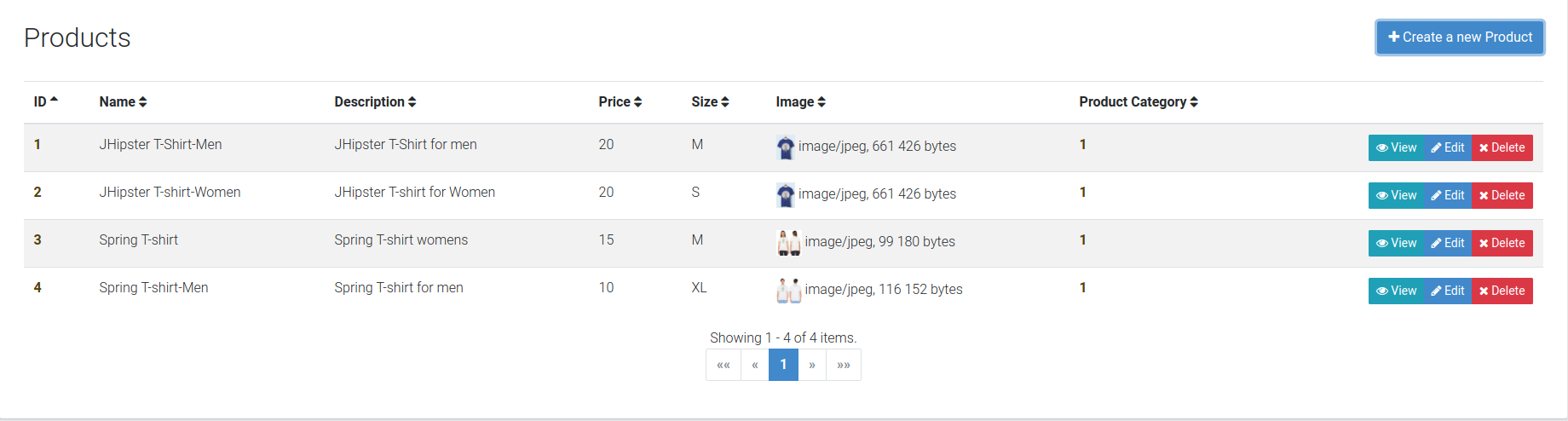
# Customizing the Angular frontend for an entity

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 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.

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 first change the view from a table into a list, but first open the development web server from BrowserSync, if it's not already open, by navigating to http://localhost:9000. Log in and navigate to Entities | Product Category and create a category, then navigate to Entities | Product and create few new products so that we have something to list:



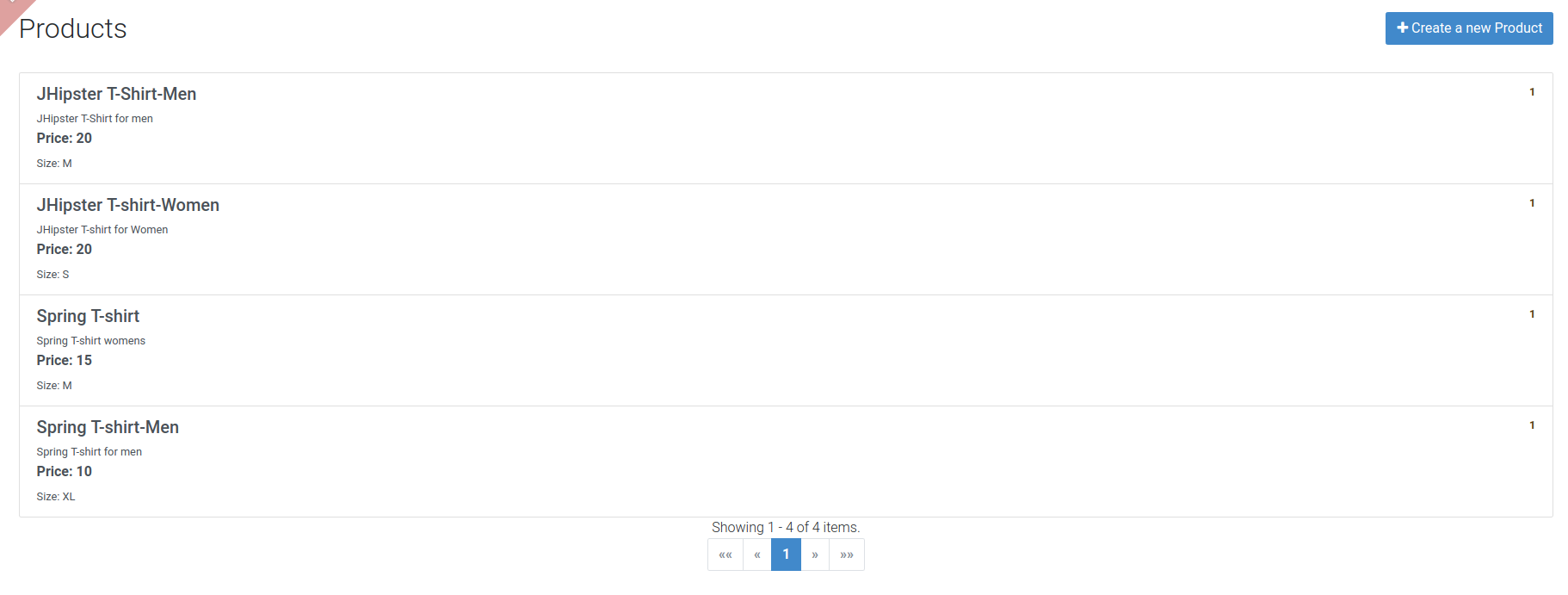
We can use the Bootstrap List group (<https://getbootstrap.com/docs/4.0/components/list-group/>) component for this purpose. Let's use the following snippet and change the view. Replace the div with class="table-responsive" with the following code:

<div **\*ngIf="products"**>  
 <div class="list-group">  
 <a **[routerLink]="['../product', product.id ]"**   
 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">  
 <h5 class="mb-1">**{{product.name}}**</h5>  
 <small \*ngIf="product.productCategory">  
 <a **[routerLink]="['../product-category',   
 product.productCategory?.id ]"** >  
 **{{product.productCategory?.id}}**  
 </a>  
 </small>  
 </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 ]" directive will create a href 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 {{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 DOM exactly, without having to recreate the entire collection. Here, we provide trackId as the function to uniquely identify an item in the collection.

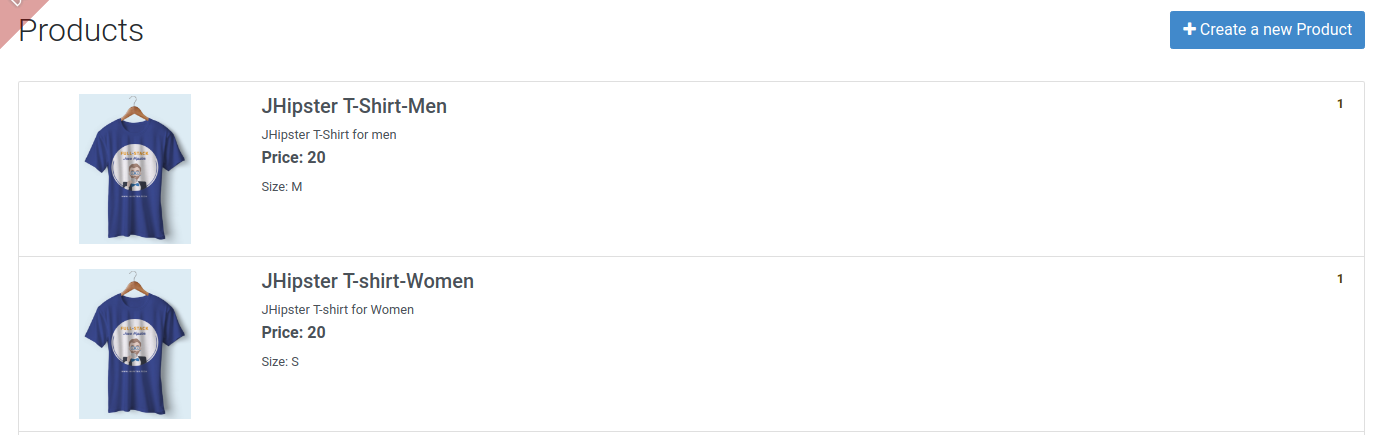
This will produce the following:



While it's a good start, it's not enough. So, let's go in and make it better. 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 rendering the content is moved into the second column and remains unchanged:

<div \*ngIf="products">  
 <div class="list-group">  
 <a [routerLink]="['../product', product.id ]" 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 class="mb-1">{{product.description}}</small>  
 <p class="mb-1">Price: {{product.price}}</p>  
 <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 divs, the first div takes up two 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 is kept responsive by specifying just col so it takes the remaining available columns after the first div, and when the display is extra small it takes up 12 columns as well. The image inside the first column div uses a data URL as src to render the image. Now we have an even better view:

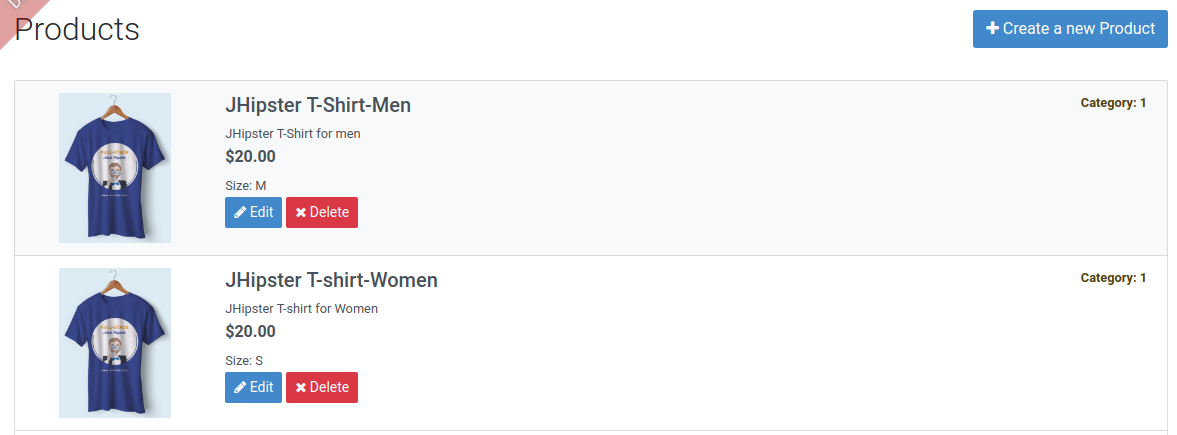


We can polish it further. We can use the Angular currency pipe (<https://angular.io/api/common/CurrencyPipe>) for the price and remove the redundant label for it by changing to {{product.price | currency:'USD'}}.  We can add a label for the category shown on the right-hand side of the list as well.

Finally, we can add the Edit and Delete buttons back, but we need to show them only for users who have the role ADMIN so that normal users will only be able to view the product listing. We can copy the HTML code for edit and delete buttons from the original table. The final code will be as follows:

<div \*ngIf="products">  
 <div class="list-group">  
 <a [routerLink]="['../product', product.id ]"   
 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">  
 <h5 class="mb-1">{{product.name}}</h5>  
 <small \*ngIf="product.productCategory">  
 <a [routerLink]="['../product-category',   
 product.productCategory?.id ]" >  
 Category: {{product.productCategory?.id}}  
 </a>  
 </small>  
 </div>  
 <small class="mb-1">{{product.description}}</small>  
 <p class="mb-1">**{{product.price | currency:'USD'}}**</p>  
 <small>  
 Size:  
 <span jhiTranslate="{{'storeApp.Size.' +   
 product.size}}">  
 {{product.size}}  
 </span>  
 </small>  
 <div **\*jhiHasAnyAuthority="'ROLE\_ADMIN'"**>  
 <button type="submit"  
 [routerLink]="['/',   
 { outlets: { popup: 'product/'+   
 product.id + '/edit'} }]"  
 replaceUrl="true"  
 queryParamsHandling="merge"  
 class="btn btn-primary btn-sm">  
 <span class="fa fa-pencil"></span>  
 <span class="d-none d-md-inline"  
 jhiTranslate="entity.action.edit">Edit</span>  
 </button>  
 <button type="submit"  
 [routerLink]="['/',   
 { outlets: { popup: 'product/'+   
 product.id + '/delete'} }]"  
 replaceUrl="true"  
 queryParamsHandling="merge"  
 class="btn btn-danger btn-sm">  
 <span class="fa fa-remove"></span>  
 <span class="d-none d-md-inline"   
 jhiTranslate="entity.action.delete">Delete</span>  
 </button>  
 </div>  
 </div>  
 </div>  
 </a>  
 </div>  
</div>

The \*jhiHasAnyAuthority="'ROLE\_ADMIN'" directive is provided by JHipster and can be used to control 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:



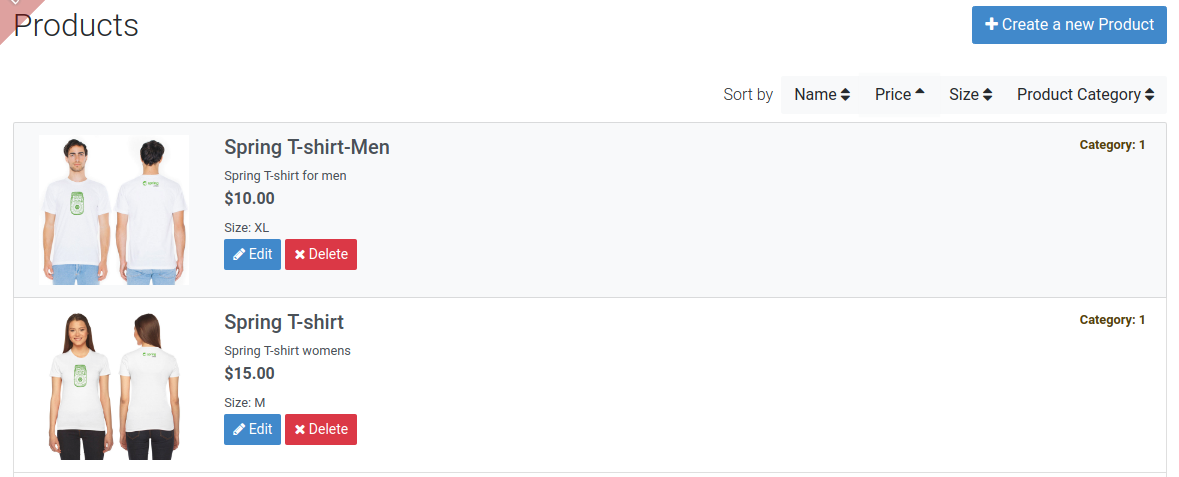
Now, let's also add the \*jhiHasAnyAuthority="'ROLE\_ADMIN'" directive to the create button element.

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

Let's use Bootstrap button group (<https://getbootstrap.com/docs/4.0/components/button-group/>) for this. Place the following snippet above the <div class="list-group"> element 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 jhiTranslate="storeApp.product.name">Name</span>  
 <span class="fa fa-sort"></span>  
 </button>  
 <button type="button" class="btn btn-light" jhiSortBy="price">  
 <span jhiTranslate="storeApp.product.price">Price</span>  
 <span class="fa fa-sort"></span>  
 </button>  
 <button type="button" class="btn btn-light" jhiSortBy="size">  
 <span jhiTranslate="storeApp.product.size">Size</span>  
 <span class="fa fa-sort"></span>  
 </button>  
 <button type="button" class="btn btn-light"   
 jhiSortBy="productCategory.id">  
 <span   
jhiTranslate="storeApp.product.productCategory">Product Category</span>  
 <span class="fa fa-sort"></span>  
 </button>  
 </div>  
</div>

We can use 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 it would use to sort. Now our page looks as follows, where products are sorted by price:



Finally, let's add some good old client-side filtering for the page.

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

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

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

Now, let's use this variable in the product.component.html file. Add the highlighted snippet from the following code to the div 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>

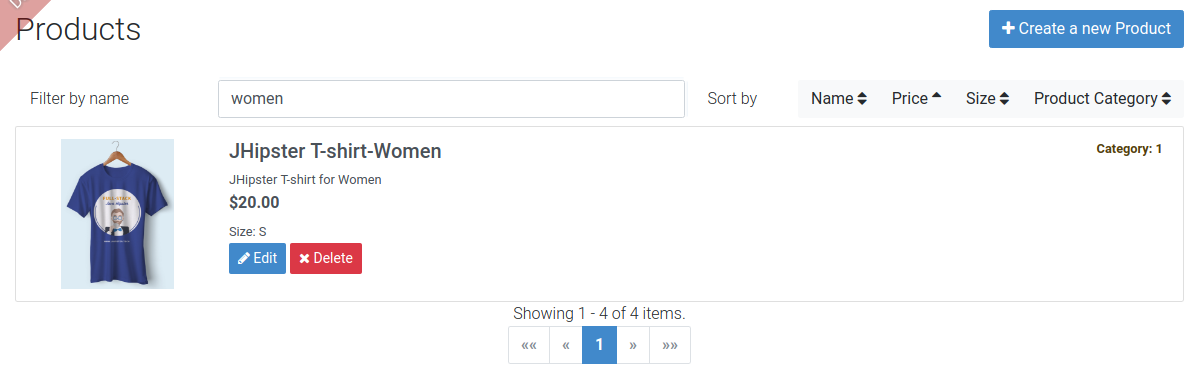
We bound the filter variable to an input element using the ngModel directive, and using [()] ensures two-way binding on the variable.

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

Finally, update the ngFor directive on our list-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, and we get a shiny filter option on our screen:



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 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"**

**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 to 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.

In our JDL, we defined relationships between 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 as follows, 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 a good practice to do 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.jh**

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

Accept the changes to the files and wait for the build to finish. Now, let's look at the entity pages to verify that the display names are used properly and create some entities to try it out. Now we realize that the Invoice entity has empty drop-down menus, 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. Now there is also another option, the entity sub generator, which can yield the same result. For the sake of familiarizing yourself with this option, let's use that to add the field to our Invoice entity:

1. Run the following command:

**> jhipster entity Invoice**

1. 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

1. 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)

1. 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.
2. Now just make sure to update the JDL so that the entity Invoice has code String required as a field.

You could also run jhipster export-jdl online-store.jh to export the current model back to the JDL.

Now that we have displayed entity relationships properly, we also need to make sure certain entities have relationship values mandatory. For example, for customers it should be mandatory to have a user, ProductOrder should have a customer, order item should have an order, Invoice should have an order, and finally, the shipment should have an invoice. Since JHipster supports making relationships required, we can make these changes using JDL. Update the relationships to the following snippet in online-store.jh:

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.jh and accept the proposed updates. Make sure to 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 use 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 codebase. 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>.

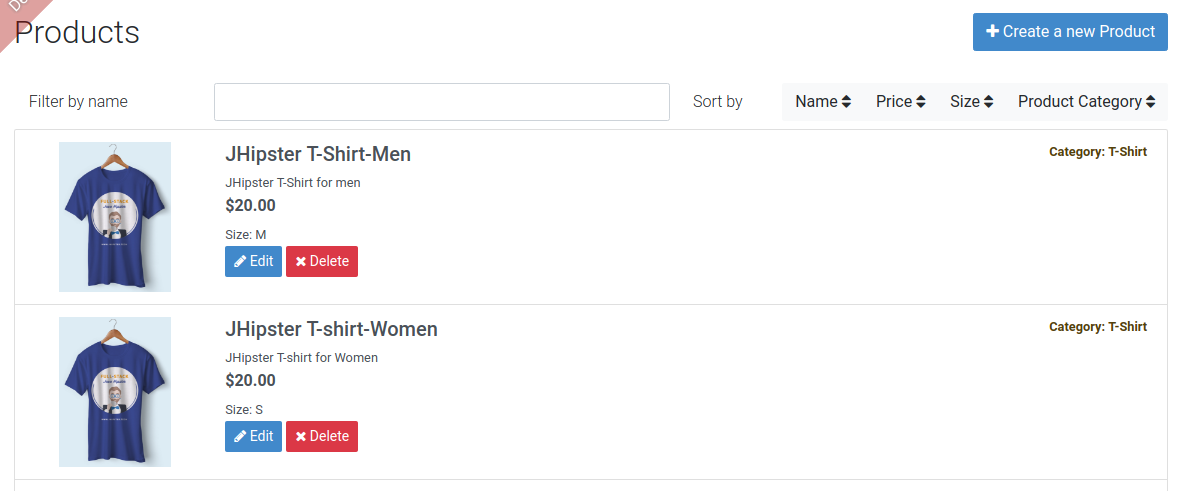
Now, this results in merge conflicts, as the product.component.html file was updated in the commit we picked on our current branch tip. We need the incoming change from the commit but also need to update the product category display name from ID to code, so let's accept the incoming change and make a 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 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**

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 VSCode, also provide good options for these.

Now our page view should be good:



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.

Since we were working on the client-side code we didn't pay attention to the server-side code that was changed during 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::20180114123500-1::jhipster was: 7:3e0637bae010a31ecb3416d07e41b621 but is now: 7:01f8e1965f0f48d255f613e7fb977628**  
 **config/liquibase/changelog/20180114123501\_added\_entity\_ProductOrder.xml::20180114123501-1::jhipster was: 7:0ff4ce77d65d6ab36f27b229b28e0cda but is now: 7:e5093e300c347aacf09284b817dc31f1**  
 **config/liquibase/changelog/20180114123502\_added\_entity\_OrderItem.xml::20180114123502-1::jhipster was: 7:2b3d9492d127add80003e2f7723903bf but is now: 7:4beb407d4411d250da2cc2f1d84dc025**  
 **config/liquibase/changelog/20180114123503\_added\_entity\_Invoice.xml::20180114123503-1::jhipster was: 7:5afaca031815e037cad23f0a0f5515d6 but is now: 7:fadec7bfabcd82dfc1ed22c0ba6c6406**  
 **config/liquibase/changelog/20180114123504\_added\_entity\_Shipment.xml::20180114123504-1::jhipster was: 7:74d9167f5da06d3dc072954b1487e11d but is now: 7:0b1b20dd4e3a38f7410b6b3c81e224fd**

This is due to the changes made to the original changelog by JHipster. In an ideal world, new schema changes should be done 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 an 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 get 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 just update the liquibaseCommand command definition in the gradle/liquibase.gradle file with the details of the production DB. Refer to <http://www.jhipster.tech/development/#using-a-database> for more.

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, then we will also 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; check it out to see all the available themes at: <https://bootswatch.com/>.

Let's use a Bootswatch theme called **materia**.

In your terminal, run yarn add 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 themes.

Now let's import this using Sass. Open src/main/webapp/content/scss/vendor.scss and find the line @import 'node\_modules/bootstrap/scss/bootstrap'; and add the following code highlighted in bold:

// Override Boostrap variables  
@import "bootstrap-variables";  
**@import 'node\_modules/bootswatch/dist/materia/variables';**// Import Bootstrap source files from node\_modules  
@import 'node\_modules/bootstrap/scss/bootstrap';  
**@import "node\_modules/bootswatch/dist/materia/bootswatch";**

The name of the theme here is materia, you can use any theme available in Bootswatch here. Make sure that name is in all lowercase. Also, notice the order of imports. It is important that we import the theme variables after importing Bootstrap variables and themes after importing the Bootstrap theme so that SASS variables and styles are overridden properly.

We can customize the theme further by overriding 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 as follows, in \_bootstrap-variables.scss:

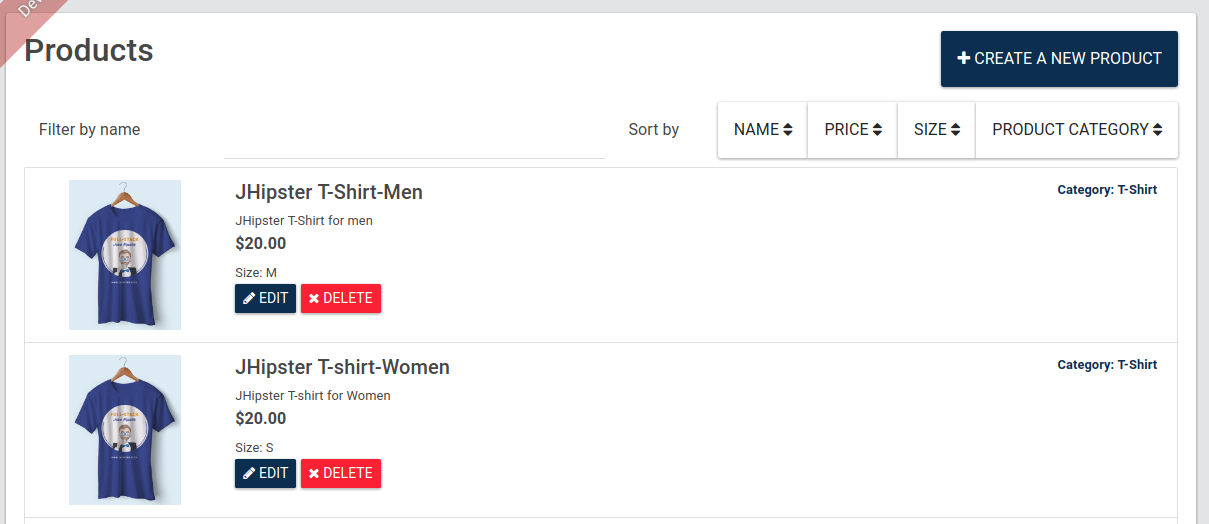
$primary: #032b4e;  
$success: #1df54f;  
$info: #17a2b8;  
$warning: #ffc107;  
$danger: #fa1a30;

There might be some UI glitches when you apply a new theme, you could solve them by updating the generated SASS files.

For example, add the following CSS to src/main/webapp/content/scss/global.scss to fix the glitch in checkboxes that we got after the theme change:

.form-check-input {  
 height: 18px;  
 width: 18px;  
}

We now have a cool new theme:



Further reference can be found at: [https://getbootstrap.com/docs/4.0/getting-started/theming/.](https://getbootstrap.com/docs/4.0/getting-started/theming/)

Let's commit this:

**> git add --all**  
**> git commit -am "update bootstrap theme using bootswatch"**

# 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.

In the terminal, switch to a new branch and run the following command:

**> git checkout -b add-french-language**  
**> jhipster languages**

You will now see a prompt like this, 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)**

Let's select French here for now. Accept the file changes proposed and we are good to go. Once the application automatically refreshes you can see the new language in the language drop-down menu in the application menu. Now, wasn't that easy!

Now there is a problem, since we have some entities and we added a new language. We will need to get i18n French files for entities as well. We can do this easily by running the jhipster --with-entities command, which will regenerate the application along with entities. Now make sure to carefully stage only 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\_fr.properties  
  
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  
  
src/main/webapp/i18n/fr

Now, let's commit this and merge it back to the master. If we have picked only i18n related changes we shouldn't have any merge conflicts:

**> git add --all**  
**> git commit -am "add French as additional language"**  
**> git merge --no-ff add-french-language**

**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 most often associated with the business logic of the application. So, it would be better if this was hand-coded by the developers as part of the business code.

Normal users have ROLE\_USER and admin users have ROLE\_ADMIN assigned in user management. For our use case there are few security holes that we need to take care of:

* Normal users should only have access to view the product listing, product order, order item, invoice, and shipment
* 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 order, order item, invoice, and shipment 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 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 elements for customer and product category 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 order item. Go to src/main/java/com/mycompany/store/web/rest/OrderItemResource.java and add @PreAuthorize("hasAuthority('ROLE\_ADMIN')") to methods createOrderItem, updateOrderItem, and deleteOrderItem:

@DeleteMapping("/order-items/{id}")  
 @Timed  
 **@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 about these can be found in the Spring Security documentation at: <https://projects.spring.io/spring-security/>.

Compile the backend using ./gradlew compileJava or using the IDE. Now go to the order items page and try to create an order item. You will get an 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 create, update, and delete methods. You could skip customer and product category entities as they are entirely forbidden to the 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 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. Let's 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, else 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 hence, 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, hence 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 docs (<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 is it. Now log in as admin and create two new users, create two customers, and create product orders for each of them. Then log out and log in again as the default user and see if you can see the product order for the newly created user.

Now let's make similar updates for other services. The repository methods for those 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 for findOne methods on the services.

For the ProductOrderService it 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.findOne(id);**  
 **} else**  
 **return productOrderRepository.findOneByIdAndCustomerUserLogin(**  
 **id,**  
 **SecurityUtils.getCurrentUserLogin().get()**  
 **);**  
 }

As you can see, we changed the methods to find one by ID and customer user login. The repository method for the same would be as follows:

ProductOrder **findOneByIdAndCustomerUserLogin**(Long id, String login);

For src/main/java/com/mycompany/store/repository/InvoiceRepository:

Invoice findOneByIdAndOrderCustomerUserLogin(Long id, String login);

For src/main/java/com/mycompany/store/repository/OrderItemRepository:

OrderItem findOneByIdAndOrderCustomerUserLogin(Long id, String login);

For src/main/java/com/mycompany/store/repository/ShipmentRepository:

Shipment findOneByIdAndInvoiceOrderCustomerUserLogin(Long id, String login);

The same queries can be written using the @Query annotation provided by Spring Data as well.

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 are not 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 it up as an assignment and see if 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 detailed in [Chapter 14](https://learning.oreilly.com/library/view/full-stack-development/9781788476317/7a7b29a3-0ff7-4d86-be16-2c4b88d3a16a.xhtml), 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.